第1章 基本概念

本章首先对C语言做简要介绍。目的是通过实际的程序向读者介绍C语言的本质要素，而不是一下子就陷入到具体细节、规则及例外情况中去。因此，在这里我们并不想完整地或很精确地对C语言进行介绍（但所举例子都是正确的）。我们想尽可能快地让读者学会编写有用的程序，因此，重点介绍其基本概念：变量与常量、算术运算、控制流、函数、基本输入输出。本章并不讨论那些编写较大的程序所需要的重要特性，包括指针、结构、大多数运算符、部分控制流语句以及标准库。

这样做也有缺陷，其中最大的不足之处是在这里找不到对任何特定语言特性的完整描述，并且，由于太简略，也可能会使读者产生误解。而且，由于所举的例子没有用到C语言的所有特性，故这些例子可能并未达到简明优美的程度。我们已尽力缩小这种差异。另一个不足之处是，本章所讲过的某些内容在后续有关章节还必须重复介绍。我们希望这种重复带给读者的帮助会胜过烦恼。

无论如何，经验丰富的程序员应能从本章所介绍的有关材料中推断他们在程序设计中需要的东西。初学者则应编写类似的小程序来充实它。这两种人都可以把本章当作了解后续各章的详细内容的框架。

1.1 入门

学习新的程序设计语言的最佳途径是编写程序。对于所有语言，编写的第一个程序都是相同的：

打印如下单词：

```
hello, world
```

在初学语言时这是一个很大的障碍，要越过这个障碍，首先必须建立程序文本，然后成功地对它进行编译，并装入、运行，最后再看看所产生的输出。只要把这些操作细节掌握了，其他内容就比较容易了。

在C语言中，用如下程序打印“hello, world”：

```
#include <stdio.h>

main()
{
   printf("hello,  world\n");
}
```

至于如何运行这个程序取决于使用的系统。作为一个特殊的例子，在UNIX操作系统中，必须首先在某个以“.c”作为扩展名的文件中建立起这个程序，如hello.c，然后再用如下命令编译

它：

```
cc hello.c
```

如果在输入上述程序时没有出现错误（例如没有漏掉字符或错拼字符），那么编译程序将往下执行并产生一个可执行文件 a.out。如果输入命令

```
a. out
```

运行a.out程序，则系统将打印

```
hello, world
```

在其他操作系统上操作步骤会有所不同，读者可向身边的专家请教。

```
#include <stdio.h>                包含有关标准库的信息

main()                            定义名为main的函数，它不接收变元值
{                                 main的语句括在花括号中
    printf("hello,  world\n");    main函数调用库函数printf打印字符序列，\n代表换行符
}
```

下面对这个程序本身做一些解释说明。每一个 C程序，不论大小如何，都由函数和变量组成。函数中包含若干用于指定所要做的计算操作的语句，而变量则用于在计算过程中存储有关值。 C中的函数类似于 FORTRAN语言中的子程序与函数或 Pascal语言中的过程与函数。在本例中，函数的名字为 main。一般而言，可以给函数任意命名，但 main是一个特殊的函数名，每一个程序都从名为main的函数的起点开始执行。这意味着每一个程序都必须包含一个 main函数。

main函数通常要调用其他函数来协助其完成某些工作，调用的函数有些是程序人员自己编写的，有些则由系统函数库提供。上述程序的第一行

```
#include <stdio.h>
```

用于告诉编译程序在本程序中包含标准输入输出库的有关信息。许多 C源程序的开始处都包含这一行。我们将在第 7章和附录 B中对标准库进行详细介绍。

在函数之间进行数据通信的一种方法是让调用函数向被调用函数提供一串叫做变元的值。函数名后面的一对圆括号用于把这一串变元（变元表）括起来。在本例子中，所定义的 main函数不要求任何变元，故用空变元表（）表示。

函数中的语句用一对花括号 {}括起来。本例中的main函数只包含一个语句：

```
printf("hello,  world\n");
```

当要调用一个函数时，先要给出这个函数的名字，再紧跟用一对圆括号括住的变元表。上面这个语句就是用变元 "hello, world\n" 来调用函数printf。printf是一个用于打印输出的库函数，在本例中，它用于打印用引号括住的字符串。

用双引号括住的字符序列叫做字符串或字符串常量，如 "hello, world\n"就是一个字符串。目前仅使用字符串作为 printf及其他函数的变元。

在C语言中，字符序列 \n表示换行符，在打印时它用于指示从下一行的左边换行打印。如果在字符串中遗漏了 \n（一个值得做的试验），那么输出打印完后没有换行。在 printf函数的变元中必须用\n引入换行符，如果用程序中的换行来代替 \n，如：

```
printf("hello,  world
```

```
");
```

那么C编译器将会产生一个错误信息。

printf函数永远不会自动换行，我们可以多次调用这个函数来分阶段打印一输出行。上面给出的第一个程序也可以写成如下形式：

```
#include <stdio.h>

main()
{
      printf("hello,  ");
      printf("world");
      printf("\n");
}
```

它所产生的输出与前面一样。

请注意，\n只表示一个字符。诸如\n等换码序列为表示不能打印或不可见字符提供了一种通用可扩充机制。除此之外， C语言提供的换码序列还有：表示制表符的 \t，表示回退符的\b，表示双引号的 \"，表示反斜杠符本身的 \\。2.3节将给出换码序列的完整列表。

练习1-1　请读者在自己的系统上运行“ hello, world ”程序。再做个实验，让程序中遗漏一些部分，看看会出现什么错误信息。

练习1-2　做个实验，观察一下当printf函数的变元字符串中包含\c（其中c是上面未列出的某个字符）时会出现什么情况。

1.2 变量与算术表达式

下面的程序用公式

$$^{\circ}C = (5/9)\,(^{\circ}F - 32)$$

打印华氏温度与摄氏温度对照表：

```
0      -17
20      -6
40       4
60      15
80      26
100     37
120     48
140     60
160     71
180     82
200     93
220    104
240    115
260    126
280    137
```

```
300    148
```

这个程序本身仍只由一个名为main的函数的定义组成，它要比前面用于打印“ hello, world ”的程序长，但并不复杂。这个程序中引入了一些新的概念，包括注解、说明、变量、算术表达式、循环以及格式输出。该程序如下：

```
#include <stdio.h>

        /* 对fahr = 0, 20, ..., 300
        打印华氏温度与摄氏温度对照表 */
        main()
        {
            int  fahr, celsius;
            int  lower, upper, step;

            lower = 0;      /* 温度表的下限 */
            upper = 300 ;   /* 温度表的上限 */
            step  = 20;     /* 步长 */

            fahr = lower;
            while  (fahr <= upper)  {
                celsius = 5 * (fahr-32) / 9;
                printf("%d\t%d\n", fahr, celsius);
                fahr = fahr + step;
            }
        }
```

其中的两行

```
/* 对fahr=0, 20, ..., 300
打印华氏温度与摄氏温度对照表 */
```

叫做注解，用于解释该程序是做什么的。夹在 /*与*/之间的字符序列在编译时被忽略掉，它们可以在程序中自由地使用，目的是为了使程序更易于理解。注解可以出现在任何空格、制表符或换行符可以出现的地方。

在C语言中，所有变量都必须先说明后使用，说明通常放在函数开始处的可执行语句之前。说明用于声明变量的性质，它由一个类型名与若干所要说明的变量组成，例如

```
int  fahr, celsius;
int  lower, upper, step;
```

其中，类型int表示所列变量为整数变量，与之相对， float表示所列变量为浮点变量（浮点数可以有小数部分）。int与float类型的取值范围取决于所使用的机器。对于 int类型，通常为16位（取值在-32768~+32767之间），也有用32位表示的。float类型一般都是32位，它至少有6位有效数字，取值范围一般在10^{-38}～10^{+38}之间。

除int与float之外，C语言还提供了其他一些基本数据类型，包括：

char	字符 —— 单字节
short	短整数

long 长整数

double 双精度浮点数

这些数据类型对象的大小也取决于机器。另外，还有由这些基本类型组成的数组、结构与联合类型、指向这些类型的指针类型以及返回这些类型的函数，我们将在后面适当的章节再分别介绍它们。

上面温度转换程序计算以4个赋值语句

```
lower = 0;
upper = 300 ;
step  = 20;
fahr = lower;
```

开始，用于为变量设置初值。各个语句均以分号结束。

温度转换表中的每一行均以相同的方式计算，故可以用循环语句来重复产生各行输出，每行重复一次。这就是while循环语句的用途：

```
while  (fahr <= upper)  {
   ...
}
```

while循环语句的执行步骤如下：首先测试圆括号中的条件。如果条件为真（ fahr小于等于upper），则执行循环体（括在花括号中的三个语句）。然后再重新测试该条件，如果为真，则再次执行该循环体。当该条件测试为假（ fahr大于upper）时，循环结束，继续执行跟在该循环语句之后的下一个语句。在本程序中，循环语句后再没有其他语句，因此整个程序终止执行。

while语句的循环体可以是用花括号括住的一个或多个语句（如上面的温度转换程序），也可以是不用花括号括住的单个语句，例如：

```
while  (i < j)
   i = 2 * i;
```

在这两种情况下，我们总是把由while控制的语句向里缩入一个制表位（在书中以四个空格表示），这样就可以很容易地看出循环语句中包含那些语句。这种缩进方式强化了程序的逻辑结构。尽管C编译程序并不关心程序的具体形式，但使程序在适当位置采用缩进空格的风格对于使程序更易于为人们阅读是很重要的。我们建议每行只写一个语句，并在运算符两边各放一个空格字符以使运算组合更清楚。花括号的位置不太重要，尽管每个人都有他所喜爱的风格。我们从一些比较流行的风格中选择了一种。读者可以选择自己所合适的风格并一直使用它。

绝大多数任务都是在循环体中做的。循环体中的赋值语句

```
celsius = 5 * (fahr-32) / 9;
```

用于求与指定华氏温度所对应的摄氏温度值并将值赋给变量 celsius。在该语句中，之所以把表达式写成先乘5然后再除以9而不直接写成5/9，是因为在C语言及其他许多语言中，整数除法要进行截取：结果中的小数部分被丢弃。由于5和9都是整数，5/9相除后所截取得的结果为0，故这样所求得的所有摄氏温度都变成0。

这个例子也对printf函数的工作功能做了更多的介绍。 printf是一个通用输出格式化函数，第7章将对此做详细介绍。该函数的第一个变元是要打印的字符串，其中百分号（ %）指示用其他

变元（第2、第3个…变元）之一对其进行替换，以及打印变元的格式。例如， %d指定一个整数变元，语句

```
printf("%d\t%d\n", fahr, celsius);
```

用于打印两个整数fahr与celsius值并在两者之间空一个制表位（\t）。

printf函数第1个变元中的各个%分别对应于第2个、第3个… 第n个变元，它们在数目和类型上都必须匹配，否则将出现错误。

顺便指出， printf函数并不是C语言本身的一部分， C语言本身没有定义输入输出功能。printf是标准库函数中一个有用的函数，标准库函数一般在 C程序中都可以使用。ANSI标准中定义了printf函数的行为，从而其性质在使用每一个符合标准的编译程序与库中都是相同的。

为了集中讨论C语言本身，在第7章之前的各章中不再对输入输出做更多的介绍，特别是把格式输入延后到第7章。如果读者想要了解数据输入，请先阅读 7.4节对scanf函数的讨论。scanf函数类似于printf函数，只不过它是用于读输入数据而不是写输出数据。

上面这个温度转换程序存在着两个问题。比较简单的一个问题是，由于所输出的数不是右对齐的，输出显得不是特别好看。这个问题比较容易解决：只要在 printf语句的第一个变元的%d中指明打印长度，则打印的数字会在打印区域内右对齐。例如，可以用

```
printf("%3d %6d\n", fahr, celsius);
```

打印fahr与celsius的值，使得fahr的值占3个数字宽、 celsius的值占6个数字宽，如下所示：

```
0        -17
20        -6
40         4
60        15
80        26
100       37
…
```

另一个较为严重的问题是，由于使用的是整数算术运算，故所求得的摄氏温度不很精确，例如，与0℉ 对应的精确的摄氏温度为-17.8℃，而不是-17℃。为了得到更精确的答案，应该用浮点算术运算来代替上面的整数算术运算。这就要求对程序做适当修改。下面给出这个程序的第二个版本：

```
#include <stdio.h>

/* 对fahr = 0, 20, ..., 300打印华氏温度与摄氏温度对照表；
   浮点数版本 */
main()
{
      float  fahr, celsius;
      int  lower, upper, step;

      lower = 0;       /* 温度表的下限 */
      upper = 300 ;    /* 温度表的上限 */
      step  = 20;      /* 步长 */
```

```
        fahr = lower;
        while  (fahr <= upper)  {
           celsius = (5.0 / 9.0) * (fahr-32.0);
           printf("%3.0f %6.1f\n", fahr, celsius);
           fahr = fahr + step;
      }
}
```

这个版本与前一个版本基本相同，只是把 fahr与celsius说明成float浮点类型，转换公式的表达也更自然。在前一个版本中，之所以不用5/9是因为按整数除法它们相除截取的结果为 0。然而，在此版本中5.0/9.0是两个浮点数相除，不需要截取。

如果某个算术运算符的运算分量均为整数类型，那么就执行整数运算。然而，如果某个算术运算符有一个浮点运算分量和一个整数运算分量，那么这个整数运算分量在开始运算之前会被转换成浮点类型。例如，对于表达式 fahr -32，32在运算过程中将被自动转换成浮点数再参与运算。不过，在写浮点常数时最好还是把它写成带小数点，即使该浮点常数取的是整数值，因为这样可以强调其浮点性质，便于人们阅读。

第2章将详细介绍把整数转换成浮点数的规则。现在请注意，赋值语句

```
fahr = lower;
```

与条件测试

```
while ( fahr <= upper )
```

也都是以自然的方式执行 —— 在运算之前先把int转换成float。

printf中的转换说明%3.0f表明要打印的浮点数（即 fahr）至少占3个字符宽，不带小数点与小数部分。%6.1f表示另一个要打印的数（ celsius）至少有6个字符宽，包括小数点和小数点后 1位数字。输出类似于如下形式：

```
  0   -17.8
 20    -6.7
 40     4.4
...
```

在格式说明中可以省去宽度（ %与小数点之间的数）与精度（小数点与字母 f之间的数）。例如，%6f的意思是要打印的数至少有 6个字符宽；%.2f说明要打印的数在小数点后有两位小数，但整个数的宽度不受限制；%f的意思仅仅是要打印的数为浮点数。

%d	打印十进制整数
%6d	打印十进制整数，至少6个字符宽
%f	打印浮点数
%6f	打印浮点数，至少6个字符宽
%.2f	打印浮点数，小数点后有两位小数
%6.2f	打印浮点数，至少6个字符宽，小数点后有两位小数

此外，printf函数还可以识别如下格式说明：表示八进制数的 %o、表示十六进制数的%x、表示字符的%c、表示字符串的%s以及表示百分号%本身的%%。

练习1-3 修改温度转换程序，使之在转换表之上打印一个标题。

练习1-4 编写一个用于打印摄氏与华氏温度对照表的程序。

1.3 for语句

对于一个特定任务，可以用多种方法来编写程序。下面是前面讲述的温度转换程序的一个变种：

```
#include <stdio.h>

/* 打印华氏与摄氏温度对照表 */
main( )
{
      int  fahr;

      for ( fahr = 0; fahr <= 300; fahr = fahr + 20 )
            printf ( "%3d   %6.1f\n", fahr,  (5.0 / 9.0) * (fahr - 32 ) );
}
```

这个版本与前一个版本执行的结果相同，但看起来有些不同。一个主要的变化是它删去了大部分变量，只留下了一个 fahr，其类型为 int。本来用变量表示的下限、上限与步长都在新引入的 for语句中作为常量出现，用于求摄氏温度的表达式现在已变成了 printf函数的第3个变元，而不再是一个独立的赋值语句。

这最后一点变化说明了一个通用规则：在所有可以使用某个类型的变量的值的地方，都可以使用该类型的更复杂的表达式。由于 printf函数的第3个变元必须为与 %6.1f匹配的浮点值，则可以在这里使用任何浮点表达式。

for语句是一种循环语句，是 while语句的推广。如果将其与前面介绍的 while语句比较，就会发现其操作要更清楚一些。在圆括号内共包含三个部分，它们之间用分号隔开。第一部分

```
fahr = 0
```

是初始化部分，仅在进入循环前执行一次。第二部分是用于控制循环的条件测试部分：

```
fahr <= 300
```

这个条件要进行求值。如果所求得的值为真，那么就执行循环体（本例循环体中只包含一个 printf函数调用语句）。然后再执行第三部分

```
fahr = fahr + 20
```

加步长，并再次对条件求值。一旦求得的条件值为假，那么就终止循环的执行。像 while语句一样，for循环语句的体可以是单个语句，也可以是用花括号括住的一组语句。初始化部分（第一部分）、条件部分（第二部分）与加步长部分（第三部分）均可以是任何表达式。

至于在while与for这两个循环语句中使用哪一个，这是随意的，主要看使用哪一个更能清楚地描述问题。 for语句比较适合描述这样的循环：初值和增量都是单个语句并且是逻辑相关的，因为for语句把循环控制语句放在一起，比 while语句更紧凑。

练习1-5 修改温度转换程序，要求以逆序打印温度转换表，即从300度到0度。

1.4 符号常量

在结束对温度转换程序的讨论之前，再来看看符号常量。把300、20等“幻数”埋在程序中并不是一种好的习惯，这些数几乎没有向以后可能要阅读该程序的人提供什么信息，而且使程序的修改变得困难。处理这种幻数的一种方法是赋予它们有意义的名字。#define指令就用于把符号名字（或称为符号常量）定义为一特定的字符串：

```
#define  名字  替换文本
```

此后，所有在程序中出现的在#define中定义的名字，该名字既没有用引号括起来，也不是其他名字的一部分，都用所对应的替换文本替换。这里的名字与普通变量名有相同的形式：它们都是以字母打头的字母或数字序列。替换文本可以是任何字符序列，而不仅限于数。

```
#include  <stdio.h>

#define  LOWER  0      /* 表的下限 */
#define  UPPER   300   /* 表的上限 */
#define  STEP    20    /* 步长 */

/* 打印华氏-摄氏温度对照表 */
main ( )
{
    int  fahr;

    for  ( fahr = LOWER; fahr <= UPPER; fahr = fahr + STEP )
         printf ( "%3d   %6.1f\n", fahr,  (5.0 / 9.0) * (fahr - 32 ) );
}
```

LOWER、UPPER与STEP等几个量是符号常量，而不是变量，故不需要出现在说明中。符号常量名通常用大写字母拼写，这样就可以很容易与用小写字母拼写的变量名相区别。注意，#define指令行的末尾没有分号。

1.5 字符输入输出

接下来讨论一些与字符数据处理有关的程序。读者将会发现，许多程序只不过是这里所讨论的程序原型的扩充版本。

由标准库提供的输入输出模型非常简单。文本的输入输出都是作为字符流处理的，不管它从何处输入、输出到何处。文本流是由一行行字符组成的字符序列，而每一行字符则由0个或多个字符组成，并后跟一个换行符。标准库有责任使每一输入输出流符合这一模型，使用标准库的C程序员不必担心各字符行在程序外面怎么表示。

标准库中有几个函数用于控制一次读写一个字符，其中最简单的是getchar和putchar这两个函数。getchar函数在被调用时从文本流中读入下一个输入字符并将其作为结果值返回。即，在执行

```
c = getchar ()
```

之后，变量c中包含了输入流中的下一个字符。这种输入字符通常是从键盘输入的。关于从文件

输入字符的方法将在第7章讨论。

putchar函数在调用时将打印一个字符。例如，函数

```
putchar ( c )
```

用于把整数变量c的内容作为一个字符打印，它通常把字符打印，通常是显示在屏幕上。putchar与printf这两个函数可以交替调用，输出的次序即调用的次序。

1.5.1 文件复制

借助getchar与putchar函数，可以在不掌握其他输入输出知识的情况下编写出许多有用的代码。最简单的程序是一次一个字符地把输入复制到输出，其基本思想如下：

```
读一个字符
while（该字符不是文件结束指示符）
    输出刚读进的字符
    读下一个字符
```

下面是其C程序：

```
#include  <stdio.h>

/* 用于将输入复制到输出的程序；第1个版本 */
main ( )
{
      int  c;

      c = getchar ( );
while ( c != EOF ) {
      putchar ( c );
      c = getchar ( );
  }
}
```

其中的关系运算符!=的意思是“不等于”。

像其他许多东西一样，一个字符不论在键盘或屏幕上以什么形式出现，在机器内部都是以位模式存储的。char类型就是专门用于存储这种字符数据的类型，当然任何整数类型也可以用于存储字符数据。由于某种微妙却很重要的理由，此处使用了int类型。

需要解决的问题是如何将文件中的有效数据与文件结束标记区分开来。C语言采取的解决方法是，getchar函数在没有输入时返回一个特殊值，这个特殊值不能与任何实际字符相混淆。这个值叫做EOF（End Of File，文件结束）。必须把c说明成一个大到足以存放getchar函数可能返回的各种值的类型。之所以不把c说明成char类型，是因为c必须大到除了能存储任何可能的字符外还要能存储文件结束符EOF。因此，把c说明成int类型的。

EOF是一个在<stdio.h>库中定义的整数，但其具体的数值是什么并不重要，只要知道它与char类型的所有值都不相同就行了。可以通过使用符号常量来保证EOF在程序中不依赖于特定的数值。

对于经验比较丰富的C程序员，可以把字符复制程序编写得更精致些。在C语言中，诸如

```
c = getchar ( )
```

之类的赋值操作是一个表达式，因而就有一个值，即赋值后位于 =左边变量的值。换言之，赋值可以作为更大的表达式的一部分出现。可以把将字符赋给 c的赋值操作放在while循环语句的测试部分中，即可以将上面的字符复制程序改写成如下形式：

```
#include  <stdio.h>

/* 用于将输入复制到输出的程序；第2个版本 */
main ( )
{
    int  c;

    while ( (c = getchar ( ) ) != EOF )
          putchar ( c );
}
```

在这一程序中，while循环语句先读一个字符并将其赋给 c，然后测试该字符是否为文件结束标记。如果该字符不是文件结束标记，那么就执行 while语句体，把该字符打印出来。再重复执行该while语句。当最后到达输入结束位置时， while循环语句终止执行，从而整个 main程序执行结束。

这个版本的特点是将输入集中处理 —— 只调用了一次 getchar函数 —— 这样使整个程序的规模有所缩短，所得到的程序更紧凑，从风格上讲，程序更易阅读。读者将会不断地看到这种风格。（然而，如果再往前走，所编写出的程序可能很难理解，我们将对这种趋势进行遏制。）

在while条件中用于括住赋值表达式的圆括号不能省略。不等运算符 !=的优先级要比赋值运算符=的优先级高，这就是说，在不使用圆括号时关系测试 !=将在赋值=之前执行。故语句

```
c = getchar ( ) != EOF
```

等价于

```
c = ( getchar ( ) != EOF )
```

这个语句的作用是把 c的值置为 0或1（取决于 getchar函数在调用执行时所读的数据是否为文件结束标记），这并不是我们所希望的结果。（有关这方面的更多的内容将在第 2章介绍。）

练习1-6　验证表达式getchar ( ) != EOF的值是0还是1。

练习1-7　编写一个用于打印EOF值的程序。

1.5.2　字符计数

下面这个程序用于对字符计数，与上面的文件复制程序类似：

```
#include  <stdio.h>

/* 统计输入的字符数； 第1个版本 */
main ( )
{
    long  nc;
```

```
    nc = 0;
    while ( getchar ( ) != EOF )
    ++nc;
    printf("%ld\n", nc);
}
```

其中的语句

```
++nc;
```

引入了一个新的运算符++，其功能是加1。可以用

```
nc = nc + 1;
```

来代替它，但++nc比之要更精致，通常效率也更高。与该运算符相对应的还有一个减1运算符--。++与--这两个运算符既可以作为前缀运算符（如++nc），也可以作为后缀运算符（如nc++）。正如第2章将要指出的，这两种形式在表达式中有不同的值，但++nc与nc++都使nc的值加1。我们暂时只使用前缀形式。

这个字符计数程序没有用int类型的变量而是用long类型的变量来存放计数值。long整数(长整数)至少要占用32位存储单元。尽管在某些机器上int与long类型的值具有同样大小，但在其他机器上int类型的值可能只有16位存储单元（最大取值32 767），相当小的输入都可能使int类型的计数变量溢出。转换说明%ld用来告诉printf函数对应的变元是long整数类型。

如果使用double(双精度浮点数)类型，那么可以统计更多的字符。下面不再用while循环语句而用for循环语句来说明编写循环的另一种方法：

```
#include  <stdio.h>

/* 统计输入的字符数； 第2个版本 */
main ( )
{
    double  nc;

    for ( nc = 0; getchar ( ) != EOF; ++nc )
        ;
    printf("%.0f\n", nc);
}
```

%f可用于float与double类型，%.0f用于控制不打印小数点和小数部分，因此小数部分为0。

这个for循环语句的体是空的，这是因为它的所有工作都在测试（条件）部分与加步长部分做了。但C语言的语法规则要求for循环语句必须有一个体，因此用单独的分号代替。单个分号叫做空语句，它正好能满足for语句的这一要求。把它单独放在一行是为了使它显目一点。

在结束讨论字符计数程序之前，请观察一下以下情况：如果输入中不包含字符，那么，while语句或for语句中的条件从一开始就为假，getchar函数一次也不会调用，程序的执行结果为0，这个结果也是正确的。这一点很重要。while语句与for语句的优点之一就是在执行循环体之前就对条件进行测试。如果没有什么事要做，那么就不去做，即使它意味着不执行循环体。程序在出现0长度的输入时应表现得机灵一点。while语句与for语句有助于保证在出现边界条件时做合理的事情。

1.5.3 行计数

下一个程序用于统计输入的行数。正如上文提到的，标准库保证输入文本流是以行序列的形式出现的，每一行均以换行符结束。因此，统计输入的行数就等价于统计换行符的个数。

```
#include  <stdio.h>

/* 统计输入的行数 */
main ( )
{
    long  c, nl;

    nl = 0;
    while ( (c = getchar ( ) ) != EOF )
          if ( c =='\n' )
    ++nl;
    printf("%d\n", nl);
}
```

这个程序中while循环语句的体是一个if语句，该if语句用于控制增值 ++nl。if语句执行时首先测试圆括号中的条件，如果该条件为真，那么就执行内嵌在其中的语句（或括在花括号中的一组语句）。这里再次用缩进方式指示哪个语句被哪个语句控制。

双等于号==是C语言中表示“等于”的运算符（类似于 Pascal中的单等于号=及FORTRN中的.EQ.）。由于C已用单等于号=作为赋值运算符，故为区别用双等于号 = =表示相等测试。注意，C语言初学者常常会用 =来表示 ==的意思。正如第 2章所述，即使这样误用了，得到的通常仍是合法的表达式，故系统不会给出警告信息。

夹在单引号中的字符表示一个整数值，这个值等于该字符在机器字符集中的数值。它叫做字符常量，尽管它只不过是比较小的整数的另一种写法。例如， 'A' 即字符常量；在ASCII字符集中其值为65（即字符A的内部表示值为65）。当然，用 'A'要比用65优越，'A'的意义清楚，并独立于特定的字符集。

字符串常量中使用的换码序列也是合法的字符常量，故 '\n' 表示换行符的值，在ASCII字符集中其值为10。我们应该仔细注意到，'\n'是单个字符，在表达式中它只不过是一个整数；而另一方面，"\n"是一个只包含一个字符的字符串常量。有关字符串与字符之间的关系将在第 2章做进一步讨论。

练习1-8　编写一个用于统计空格、制表符与换行符个数的程序。

练习1-9　编写一个程序，把它的输入复制到输出，并在此过程中将相连的多个空格用一个空格代替。

练习1-10　编写一个程序，把它的输入复制到输出，并在此过程中把制表符换成 \t、把回退符换成\b、把反斜杠换成\\。这样可以使得制表符与回退符能以无歧义的方式可见。

1.5.4 单词计数

我们将要介绍的第四个实用程序用于统计行数、单词数与字符数，这里对单词的定义放得

比较宽，它是任何其中不包含空格、制表符或换行符的字符序列。下面这个比较简单的版本是在UNIX系统上实现的完成这一功能的程序 wc：

```
#include  <stdio.h>

#define IN   1  /* 在单词内 */
#define OUT  0  /* 在单词外 */

/* 统计输入的行数、单词数与字符数 */
main ( )
{
    int c, nl, nw, nc, state;

    state = OUT;
    nl = nw = nc = 0;
    while ( (c = getchar ( ) ) != EOF ) {
        ++nc;
        if ( c =='\n' )
            ++nl;
        if ( c == ' ' ||c == '\n' || c == '\t' )
            state = OUT;
        else if ( state == OUT ) {
                 state = IN;
                 ++nw;
                      }
            }
            printf("%d  %d  %d\n", nl,  nw, nc);
}
```

程序在执行时，每当遇到单词的第一个字符，它就作为一个新单词加以统计。 state变量用于记录程序是否正在处理一个单词（是否在一个单词中），它的初值是“ 不在单词中 ”，即被赋初值为OUT。我们在这里使用了符号常量 IN与OUT而没有使用其字面值 1与0，主要是因为这可以使程序更可读。在比较小的程序中，这样做也许看不出有什么区别，但在比较大的程序中，如果从一开始就这样做，那么所增加的一点工作量与所提高的程序的明晰性相比是很值得的。读者也会发现，在程序中，如果幻数仅仅出现在符号常量中，那么对程序做大量修改就显得比较容易。

语句行

```
nl = nw = nc = 0;
```

用于把其中的三个变量nl、nw 与nc都置为0。这种情况并不特殊，但要注意这样一个事实，在兼有值与赋值两种功能的表达式中，赋值结合次序是由右至左。所以上面这个语句也可以写成：

```
nl = ( nw = ( nc = 0 ));
```

运算符 || 的意思是OR（或），所以程序行

```
if ( c == ' ' ||c == '\n' || c == '\t' )
```

的意思是“ 如果c是空格或c是换行符或c是制表符 ”（回忆一下，换码序列 \t 是制表符的可见表

示)。与之对应的一个运算符是&&，其含义是AND（与），其优先级只比||高一级。经由&&或||连接的表达式由左至右求值，并保证在求值过程中只要已得知真或假，求值就停止。如果c是一个空格，那么就没有必要再测试它是否为换行符或制表符，故后两个测试无需再进行。在这里这倒不特别重要，但在某些更复杂的情况下这样做就显得很重要，不久我们将会看到这种例子。

这个例子中还给出了else部分，它指定当if语句中的条件部分为假时所要采取的动作。其一般形式为：

```
if  (表达式)
        语句1
else
        语句2
```

在if-else的两个语句中有一个并且只有一个被执行。如果表达式的值为真，那么就执行语句$_1$，否则，执行语句$_2$。这两个语句均可以或者是单个语句或者是括在花括号内的语句序列。在单词计数程序中，else之后的语句仍是一个if语句，这个if语句用于控制括在花括号中的两个语句。

练习1-11　你准备怎样测试单词计数程序？如果程序中出现任何错误，那么什么样的输入最有利于发现这些错误？

练习1-12　编写一个程序，以每行一个单词的形式打印输入。

1.6 数组

下面编写一个用来统计各个数字、空白字符（空格符、制表符及换行符）以及所有其他字符出现次数的程序。这个程序听起来有点矫揉造作，但有助于在一个程序中对C语言的几个方面加以讨论。

由于所有输入的字符可以分成12个范畴，因此用一个数组比用十个独立的变量来存放各个数字的出现次数要方便一些。下面是这个程序的第一个版本：

```
#include  <stdio.h>
/* 统计各个数字、空白字符及其他字符分别出现的次数 */
main ( )
{
    int  c, i, nwhite, nother;
    int  ndigit[10];

    nwhite = nother = 0;
    for ( i =0;  i < 10; ++i )
        ndigit[i] = 0;

    while ( ( c = getchar() ) != EOF )
        if ( c >= '0' && c <= '9' )
            ++ndigit[c - '0'];
        else if ( c == ' ' ||c == '\n' || c == '\t' )
            ++nwhite;
        else
```

```
            ++nother;

    printf( "digits =" );
    for ( i =0; i < 10; ++i )
        printf( "  %d", ndigit[i] );
        printf( ", white space = %d, other = %d", nwhite, nother);
}
```

当把程序自身作为输入时，输出为：

```
digits = 9 3 0 0 0 0 0 0 0 1, white space = 123, other = 345
```

程序中的说明语句

```
int  ndigit[10];
```

用于把ndigit说明为由10个整数组成的数组。在C语言中，数组下标总是从0开始，故这个数组的10个元素是ndigit[0]、ndigit[1]、…、ndigit[9]。这一点在分别用于初始化和打印数组的两个 for 循环语句中得到反映。

下标可以是任何整数表达式，包括整数变量（如 i）与整数常量。

这个程序的执行取决于数字的字符表示的性质。例如，测试

```
if ( c >= '0' && c <= '9' ) ...
```

用于判断c中的字符是否为数字。如果它是数字，那么该数字的数值是：

```
c - '0'
```

这种做法只有在'0'、'1'、…、'9'具有连续增加的值时才有效。所幸所有字符集都是这样做的。

根据定义，char类型的字符是小整数，故char类型的变量和常量等价于算术表达式中 int类型的变量和常量。这样做既自然又方便，例如， c - '0'是一个整数表达式，对应于存储在c中的字符'0'至'9'，其值为0至9，因此可以充当数组ndigit的合法下标。

关于一个字符是数字、空白字符还是其他字符的判定是由如下语句序列完成的：

```
if ( c >= '0' && c <= '9' )
    ++ndigit[c - '0'];
else if ( c == ' ' ||c == '\n' || c == '\t' )
    ++nwhite;
else
    ++nother;
```

在程序中经常会用如下模式来表示多路判定：

```
if (条件1)
    语句1
else if (条件2)
    语句2
…
    …
else
    语句n
```

在这个模式中，各个条件从前往后依次求值，直到满足某个条件，这时执行对应的语句部

分，语句执行完成后，整个if构造完结。（其中的任何语句都可以是括在花括号中的若干个语句。）如果其中没有一个条件满足，那么就执行位于最后一个 else之后的语句（如果有这个语句）。如果没有最后一个else及对应的语句，那么这个 if构造就不执行任何动作，如同前面的单词计数程序一样。在第一个if与最后一个else之间可以有0个或多个

```
else if (条件)
    语句
```

就风格而言，我们建议读者采用缩进格式。如果每一个 if都比前一个else向里缩进一点，那么对一个比较长的判定序列就有可能越出页面的右边界。

第3章将要讨论的switch语句提供了编写多路分支的另一种手段，它特别适合于表示数个整数或字符表达式是否与一常量集中的某个元素匹配的情况。为便于对比，我们将在 3.4节给出用switch语句编写的这个程序的另一个版本。

练习1-13 编写一个程序，打印其输入的文件中单词长度的直方图。横条的直方图比较容易绘制，竖条直方图则要困难些。

练习1-14 编写一个程序，打印其输入的文件中各个字符出现频率的直方图。

1.7 函数

C语言中的函数类似于 FORTRAN语言中的子程序或函数，或者 Pascal语言中的过程或函数。函数为计算的封装提供了一种简便的方法，在其他地方使用函数时不需要考虑它是如何实现的。在使用正确设计的函数时不需要考虑它是怎么做的，只需要知道它是做什么的就够了。 C语言使用了简单、方便、有效的函数，我们将会经常看到一些只定义和调用了一次的短函数，这样使用函数使某些代码段更易于理解。

到目前为此，我们所使用的函数（如 printf、getchar与putchar等）都是函数库为我们提供的。现在是我们自己编写一些函数的时候了。由于 C语言没有像FORTRAN语言那样提供诸如 **之类的乘幂运算符，我们可以通过编写一个求北幂的函数 power(m, n)来说明定义函数的方法。power(m, n)函数用于计算整数 m的正整数次幂 n，即power(2,5)的值为32。这个函数不是一个实用的乘幂函数，它只能用于处理比较小的整数的正整数次幂，但它对于说明问题已足够了。（在标准库中包含了一个用于计算 xy的函数pow(x, y)。）

下面给出函数power(m, n)的定义及调用它的主程序，由此可以看到整个结构。

```
#include  <stdio.h>

int  power(int m, int n);

/* 测试power函数 */
main ( )
{
    int  i;

    for ( i =0; i < 10; ++i )
```

```
        printf("%d  %d  %d\n", i, power(2, i), power(-3, i));
    return 0;
}

/* power: 求底的n次幂； n >=0  */
int  power(int base, int n)
{
    int  i, p;

    p = 1;
    for ( i = 1; i <= n; ++i )
        p = p * base;
    return p;
}
```

函数定义的一般形式为：

```
返回值类型函数名（可能有的参数说明）
{
    说明序列
    语句序列
}
```

不同函数的定义可以以任意次序出现在一个源文件或多个源文件中，但同一函数不能分开存放在几个文件中。如果源程序出现在几个文件中，那么对它的编译和装入比将整个源程序放在同一文件时要做更多说明，但这是操作系统的任务，而不是语言属性。我们暂且假定两个函数放在同一文件中，从而前面所学的有关运行 C程序的知识在目前仍然有用。

main主程序在如下命令行中对power函数进行了两次调用：

```
printf("%d  %d  %d\n", i, power(2, i), power(-3, i));
```

每一次调用均向power函数传送两个变元，而 power函数则在每次调用执行完时返回一个要按一定格式打印的整数。在表达式中， power(2, i)就像２和i一样是一个整数。（并不是所有函数都产生一个整数值，第4章将对此进行讨论。）

power函数本身的第一行

```
int  power(int base, int n)
```

说明参数的类型与名字以及该函数返回的结果的类型。 power的参数名只能在 power内部使用，在其他函数中不可见：在其他函数中可以使用与之相同的参数名而不会发生冲突。对变量 i与p亦如此：power函数中的i与main函数中的i无关。

一般而言，把在函数定义中用圆括号括住的表中命名的变量叫做参数，而把函数调用中与参数对应的值叫做变元。为了表示两者的区别，有时也用形式变元与实际变元这两个术语。

power函数计算得的值由return语句返回给main函数。关键词return可以后跟任何表达式：

```
return 表达式；
```

函数不一定都返回一个值。不含表达式的 return语句用于使控制返回调用者（但不返回有用的值），如同在达到函数的终结右花括号时“脱离函数”一样。调用函数也可以忽略（不用）一

个函数所返回的值。

读者可能已经注意到，在main函数末尾有一个return语句。由于main本身也是一个函数，它也就可以向其调用者返回一个值，这个调用者实际上就是程序的执行环境。一般而言，返回值为0表示正常返回，返回值非 0则引发异常或错误终止条件。从简明性角度考虑，在这之前的main函数中都省去了return语句，但在以后的main函数中将包含return语句，以提醒程序要向环境返回状态。

main函数前的说明语句

```
int  power(int m, int n);
```

表明power是一个有两个int类型的变元并返回一个int类型的值的函数。这个说明叫做函数原型，要与power函数的定义和使用相一致。如果该函数的定义和使用与这一函数原型不一致，那么就是错误的。

函数原型与函数说明中参数的名字不要求相同。更确切地说，函数原型中的参数名是可有可无的。故上面这个函数原型也可以写成：

```
int  power(int, int);
```

但是，选择一个合适的参数名是一种良好的文档编写风格，我们在使用函数原型时仍将指明参数名。

历史回顾：ANSI C和C的较早版本之间的最大区别在于函数的说明与定义方法。在 C语言的最初定义中，power函数要写成如下形式：

```
/* power: 求底的n次幂；n >=0  */
/*  (老风格版本)  */
power(base, n)
int base, n;
{
    int  i, p;

    p = 1;
    for ( i = 1; i <= n; ++i )
        p = p * base;
    return p;
}
```

参数的名字在圆括号中指定，但参数类型则在左花括号之前说明，如果一个参数未在这一位置加以说明，那么缺省为int类型。(函数的体与ANSI C相同。)

在程序的开始处，也可以将power说明成如下形式：

```
int  power( );
```

在函数说明中不包含参数，这样编译程序就不能马上对调用 power的合法性进行检查。实际上，由于在缺省情况下假定power要返回int类型的值，这个函数说明可以全部省去。

在新定义的函数原型的语法中，编译程序可以很容易检测函数调用在变元数目和类型方面的错误。在ANSI C中仍可以使用老风格的函数说明与定义，至少它可以作为一个过渡阶段。但我们还是强烈建议读者：在可以使用支持新风格的编译程序时，最好使用新形式的函数原型。

练习1-15　重写1.2节的温度转换程序，使用函数来实现温度转换。

1.8　变元 —— 按值调用

使用其他语言（特别是FORTRAN语言）的程序员可能对C语言有关参数的这样一个方面不太熟悉。在C语言中，所有函数变元都是“按值”传递的。这意味着，被调用函数所得到的变元值放在临时变量中而不是放在原来的变量中。这样它的性质就与诸如FORTRAN等采用“按引用调用”的语言或诸如Pascal等采用var参数的语言有所不同，在这些语言中，被调用函数必须访问原来的变元，而不是采用局部复制的方法。

最主要的区别在于，在C语言中，被调用函数不能直接更改调用函数中变量的值，它只能更改其私有临时拷贝的值。

按值调用有益处而非弊端。由于在采用按值调用时在被调用函数中参数可以像通常的局部变量一样处理，这样可以使函数中只使用少量的外部变量，从而使程序更简洁。例如，下面是利用这一性质的power版本：

```
/* power: 求底的n次幂； n >=0；第2个版本  */
int  power(int base, int n)
{
    int  p;

    for ( n = 1; n > 0; --n )
        p = p * base;
    return p;
}
```

参数n被用作临时变量，（通过一个向后执行的for循环语句）向下计数，一直到其值变成0，这样就不再需要引入变量i。在power函数内部对n的操作不会影响到调用函数在调用power时所使用的变元的值。

在必要时，也可以在对函数改写，使之可以修改调用例程中的变量。此时调用者要向被调用函数提供所要改变值的变量的地址（从技术角度看，地址就是指向变量的指针），而被调用函数则要把对应的参数说明成指针类型，并通过它间接访问变量。我们将在第5章讨论指针。

对数组的情况有所不同。当把数组名用作变元时，传递给函数的值是数组开始处的位置或地址 —— 不是数组元素的副本。在被调用函数中可以通过数组下标来访问或改变数组元素的值。这是下一节所要讨论的问题。

1.9　字符数组

C语言中最常用的数组类型是字符数组。为了说明字符数组以及用于处理字符数组的函数的用法，我们来编写一个程序，它用于读入一组文本行并把最长的文本行打印出来。对其算法描述相当简单：

```
while  (还有没有处理的行)
   if  (该行比已处理的最长的行还要长)
       保存该行
```

保存该行的长度
打印最长的行

这一算法描述很清楚，很自然地把所要编写的程序分成了若干部分，分别用于读入新行、测试读入的行、保存该行及控制这一过程。

由于分割得比较好，故可以像这样来编写程序。首先编写一个独立的函数 getline来读取输入的下一行。我们想使这个函数在其他地方也能使用。 getline函数至少在读到文件末尾时要返回一个信号，而更有用的设计是它能在读入文本行时返回该行的长度，而在遇到文件结束符时返回 0。由于0不是有效的行长度，因此是一个可以接受的标记文件结束的返回值。每一行至少要有一个字符，只包含换行符的行的长度为 1。

当发现某一个新读入的行比以前读入的最长的行还要长时，就要把该新行保存起来。这意味着需要用第二个函数copy来把新行复制到一个安全的位置。

最后，需要用主函数main来控制对getline和copy这两个函数的调用。整个程序如下：

```
#include  <stdio.h>
#define  MAXLINE  1000    /* 最大输入行的大小 */

int  getline (char line[ ], int maxline );
void  copy ( char to[ ], char from [] );

/* 打印最长的输入行 */
main ( )
{
    int  len;                /* 当前行长度 */
    int  max;                /* 至目前为止所发现的最长行的长度 */
    char  line[MAXLINE];     /* 当前输入的行 */
    char  longest[MAXLINE];  /* 用于保存最长的行 */

    max = 0;
    while  ( ( len = getline (line, MAXLINE) ) > 0 )
        if  (len > max)  {
            max = len;
            copy ( longest, line );
        }
    if  (max > 0)    /* 有一行 */
        printf ("%s" , longest ) ;
    return 0 ;
}

/* getline：将一行读入s中并返回其长度 */
int getline (char s [ ], int lim)
{
    int  c, i;

    for  (i = 0; i < lim -1 && (c = getchar ( ) ) != EOF && c != '\n'; ++i )
         s[i] = c;
```

```
    if   (c == '\n' )  {
         s[i] = c;
         ++i;
    }
    s[i] =  '\0';
    return i;
}

/* copy：从from拷贝到to; 假定to足够大 */
void copy ( char to [ ], char from [ ])
{
    int i;

    i = 0;
    while  ( ( to[ i ] = from [ i ]) != '\0')
       ++i;
}
```

在程序的一开始就对getline和copy这两个函数进行了说明，假定它们都放在同一文件中。

main与getline这两个函数通过一对变元及一个返回值进行交换。在getline函数中，两个变元是通过程序行

```
int getline (char s [ ], int lim)
```

说明的，它把第一个变元s说明成数组，把第二个变元lim说明成整数。在说明中提供数组大小的目的是留出存储空间。在getline函数中没有必要说明数组s的长度，因为该数组的大小是在main函数中设置的。如同power函数一样，getline函数使用了一个return语句把值回送给其调用者。这一程序行也说明了getline函数的返回值类型为int，由于int为缺省返回值类型，故可以省略。

有些函数要返回一个有用的值，而另外一些函数（如 copy）则仅用于执行一些动作，并不返回值。copy函数的返回类型为void，它用于显式指明该函数不返回任何值。

getline函数把字符 '\0'（即空字符，其值为0）放到它所建立的数组的末尾，以标记字符串的结束。这一约定也已被C语言采用，当在C程序中出现诸如

```
"hello\n"
```

的字符串常量时，它被作为字符数组存储，该数组中包含这个字符串中各个字符并以 '\0' 来标记字符串结束：

h	e	l	l	o	\n	\0

在printf库函数中，格式说明%s要求所对应的变元是以这种形式表示的字符串。 copy函数也基于这样的事实，其输入变元值以 '\0' 结束，并将这个字符拷贝到输出变元中。（所有这一切都意味着空字符 '\0' 不是通常文本的一部分。）

值得一提的是，在传递参数（变元）时，即使是像本例这样很小的程序也会遇到某些麻烦的设计问题。例如，当所遇到的行比所允许的最大值还要大时， main函数应该怎么处理？getline函数的执行是安全的，当数组满了时它就停止读字符，即使它还没有遇到换行符。 main函数可以通过测试行长度以及所返回的最后一个字符来判定该行是否太长，然后再按其需要进

行处理。为了使程序简洁一些，我们在这里将不处理这个问题。

getline函数的调用程序没有办法预先知道一个输入行可能有多长，故getline函数采用了检查溢出的方法。另一方面，copy函数的调用程序则已经知道（或可以找出）所处理字符串的长度，故其中没有错误检查处理。

练习1-16 对用于打印最长行的程序的主程序main进行修改，使之可以打印任意长度的输入行的长度以及文本行中尽可能多的字符。

练习1-17 编写一个程序，把所有长度大于80个字符的输入行都打印出来。

练习1-18 编写一个程序，把每个输入行中的尾部空格及制表符都删除掉，并删除空格行。

练习1-19 编写函数reverse(s)，把字符串s颠倒过来。用它编写一个程序，一次把一个输入行字符串颠倒过来。

1.10 外部变量与作用域

main函数中的变量（如line、longest等）是main函数私有的或称局部于main函数的。由于它们是在main函数中说明的，其他函数不能直接访问它们。在其他函数中说明的变量也同样如此，例如，getline函数中说明的变量i与copy函数中说明的变量i没有关系。函数中的每一个局部变量只在该函数被调用时存在，在该函数执行完退出时消失。这也是仿照其他语言通常把这类变量称为自动变量的原因。以后我们将用自动变量来指局部变量。（第4章将讨论静态存储类，属于静态存储类的局部变量在函数调用之间保持其值不变。）

由于自动变量只在函数调用执行期间存在，故在函数的两次调用期间自动变量不保留在前次调用时所赋的值，且在函数的每次调用执行时都要显式给其赋初值。如果没有给自动变量赋初值，那么其中所存放的是无用数据。

作为对自动变量的替补，可以定义适用于所有函数的外部变量，即可以被所有函数通过变量名访问的变量。（这一机制非常类似于FORTRAN语言的COMMON变量或Pascal语言在最外层分程序中说明的变量。）由于外部变量可以全局访问，因此可以用外部变量代替变元表用于在函数间交换数据。而且，外部变量在程序执行期间一直存在，而不是在函数调用时产生、在函数执行完时消失，即使从为其赋值的函数返回后仍保留原来的值不变。

外部变量必须在所有函数之外定义，且只能定义一次，定义的目的是为之分配存储单元。在每一个函数中都要对所要访问的外部变量进行说明，说明所使用外部变量的类型。在说明时可以用extern语句显式指明，也可以通过上下文隐式说明。为了更具体地讨论外部变量，我们重写上面用于打印最长行的程序，把line、longest与max说明成外部变量。这需要修改所有这三个函数的调用、说明与函数体。

```
#include  <stdio.h>

#define  MAXLINE  1000    /* 最大输入行的大小 */

int  max; /* 至目前为止所发现的最长行的长度 */
char  line[MAXLINE];      /* 当前输入的行 */
```

```
char  longest[MAXLINE];   /* 用于保存最长的行 */

int  getline (void );
void  copy ( void );

/* 打印最长的输入行； 特别版本 */
main ( )
{
    int  len;
    extern int  max;
    extern char  longest[ ];

    max = 0;
    while  ( ( len = getline ( ) ) > 0 )
        if  (len > max)  {
            max = len;
            copy ( );
        }
    if  (max > 0)    /* 有一行 */
        printf ("%s" , longest ) ;
    return 0 ;
}

/* getline：特别版本 */
int getline (void )
{
    int  c, i;
    extern char  line[ ];

    for  (i = 0; i < MAXLINE -1 && (c = getchar ( ) ) != EOF && c != '\n'; ++i )
         line[i] = c;
    if  (c == '\n' )  {
        line[i] = c;
        ++i;
    }
    line[i] = '\0';
    return i;
}

/* copy：特别版本 */
void copy ( void )
{
    int i;
    extern char  line[ ], longest[ ];

    i = 0;
    while  ( ( longest[ i ] = line [ i ]) != '\0')
```

```
            ++i;
    }
```

在这个例子中，前几行定义了在main、getline与copy函数中使用的几个外部变量，指明各外部变量的类型并使系统为之分配存储单元。从语法角度看，外部变量定义就像局部变量的定义一样，但由于它们出现在各个函数的外部，这些变量就成了外部变量。一个函数在使用外部变量之前必须使变量的名字在该函数中可见，一种方法是在该函数中编写一个extern说明，说明除了在前面加了一个关键词extern外，其他地方均与普通说明相同。

在某些情况下，extern说明可以省略。如果外部变量的定义在源文件中出现在使用它的函数之前，那么在该函数中就没有必要使用extern说明。于是，main、getline及copy中的几个extern说明都是多余的。事实上，比较常用的做法是把所有外部变量的定义放在源文件的开始处，这样就可以省略extern说明。

如果程序包含几个源文件，某个变量在file1文件中定义、在file2与file3文件中使用，那么在file2与file3文件中就需要使用extern说明来连接该变量的出现。人们通常把变量的extern说明与函数放在一个单独的文件中（历史上叫做头文件），在每一个源文件的前面用#include语句把所要用的头文件包含进来。后缀.h被约定为头文件名的扩展名。例如，标准库中的函数就是在诸如<stdio.h>的头文件中说明的。这一问题将在第4章详细讨论，而库本身在第7章及附录B中讨论。

由于getline与copy函数的特别版本中不带有变元，从道理上讲，在源文件开始处它们的原型应该是getline( )与copy( )。但为了与较老的C程序相兼容，C标准把空变元表作为老式说明，并关闭所有对变元表的检查。如果变元表本身是空的，那么要使用关键词void，第4章将对此做进一步讨论。

读者应该注意到，这一节我们在说到外部变量时很小心谨慎地使用着两个词定义与说明。“定义”指变量建立或分配存储单元的位置，而“说明”则指指明变量性质的位置，但并不分配存储单元。

顺便指出，现在有一种把所有看得见的东西都作为外部变量的趋势，因为这样似乎可以简化通信——变元表变短了，且变量在需要时总是存在。但外部变量即使在不需要时也还是存在的。过分依赖于外部变量充满了危险，因为这将会使程序中的数据联系变得很不明显——外部变量的值可能会被意外地或不经意地改变，程序也变得难以修改。上面打印最长行的程序的第2个版本就不如第1个版本，之所以如此，部分是由于这个原因，部分是由于它把两个有用的函数所操纵的变量的名字绑到函数中，使这两个函数失去了一般性。

到目前为此，我们已经对C语言传统的核心部分进行了介绍。借助于这少量的构件，我们已经能编写出相当规模的程序，因此建议读者花上较长的时间来编写一些程序。下面给出的练习比本章前面的程序复杂一些。

练习1-20　编写程序detab，将输入中的制表符替换成适当数目的空白符（使空白充满到下一制表符停止位）。假定制表符停止位的位置是固定的，比如在每个n列的位置上。n应作成变量或符号参数吗？

练习1-21　编写程序entab，将空白符串用可达到相同空白的最小数目的制表符和空白符来

替换。使用与detab程序相同的制表停止位。请问，当一个制表符与一个空白符都可以到达制表符停止位时，选用哪一个比较好？

练习1-22　编写一个程序，用于把较长的输入行“折”成短一些的两行或多行，折行的位置在输入的第n列之前的最后一个非空白字符之后。要保证程序具备一定的智能，能应付输入行很长以及在指定的列前没有空白符或制表符时的情况。

练习1-23　编写一个用于把C程序中所有注解都删除掉的程序。不要忘记处理好带引号的字符串与字符常量。在C程序中注解不允许嵌套。

练习1-24　编写一个程序，查找C程序中的基本语法错误，如圆括号、方括号、花括号不配对等。不要忘记引号（包括单引号和双引号）、换码序列与注解。（如果读者想把该程序编写成完全通用性的，那么难度比较大。）

第2章　类型、运算符与表达式

变量与常量是程序中所要处理的两种基本数据对象。说明语句中列出了所要使用的变量的名字及该变量的类型，可能还要给出该变量的初值。运算符用于指定要对变量与常量进行的操作。表达式则用于把变量与常量组合起来产生新的值。一个对象的类型决定着该对象可取值的集合以及可以对该对象施行的运算。本章将要对这些构件进行详细讨论。

ANSI C语言标准对语言的基本类型与表达式做了许多小的修改与增补。所有整数类型现在都有signed（有符号）与unsigned（无符号）两种形式，且可以表示无符号常量与十六进制字符常量。浮点运算可以以单精度进行，另外还可以使用更高精度的 long double类型。字符串常量可以在编译时连接。枚举现在也成了语言的一部分，这是经过长期努力才形成的语言特征。对象可以说明成const（常量），这种对象的值不能进行修改。语言还对算术类型之间的自动强制转换规则做了扩充，使这一规则可以适合更多的数据类型。

2.1　变量名

对变量与符号常量的名字存在着一些限制，这一点在第 1章中没有指出来。名字由字母与数字组成，但其第一个字符必须为字母。下划线_也被看做是字母，它有时可用于命名比较长的变量名以提高可读性。由于库函数通常使用以下划线开头的名字，因此不要将这类名字用做变量名。大写字母与小写字母是有区别的， x与X是两个不同的名字，一般把由大写字母组成的名字用做符号常量。

在内部名字中至少前 31个字符是有效的。对于函数名与外部变量名，其中所包含的字符的数目可以小于31个，这是因为它们可能会被语言无法控制的汇编程序和装配程序使用。对于外部名，ANSI C标准保证了唯一性仅对前 6个字符而言并且不区分大小写。诸如 if、else、int、float等关键词是保留的，不能把它们用做变量名。所有关键词中的字符都必须小写。

在选择变量名时比较明智的方法是使所选名字的含义能表达变量的用途。我们倾向于局部变量使用比较短的名字（尤其是循环控制变量，亦叫循环位标），外部变量使用比较长的名字。

2.2　数据类型与大小

在C语言中只有如下几个基本数据类型：

char	单字节，可以存放字符集中一个字符。
int	整数，一般反映了宿主机上整数的自然大小。
float	单精度浮点数。
double	双精度浮点数。

此外，还有一些可用于限定这些基本类型的限定符。其中 short与long这两个限定符用于限定整数类型：

```
short int  sh;
long int  counter;
```

在这种说明中，int可以省去，一般情况下许多人也是这么做的。

引入这两个类型限定符的目的是为了使 short与long提供各种满足实际要求的不同长度的整数。int通常反映特定机器的自然大小，一般为 16位或32位，short对象一般为16位，long对象一般为32位。各个编译程序可以根据其硬件自由选择适当的大小，唯一的限制是， short与int对象至少要有16位，而long对象至少要有 32位；short对象不得长于 int对象，而int对象则不得长于long对象。

类型限定符signed与unsigned可用于限定char类型或任何整数类型。经unsigned限定符限定的数总是正的或0，并服从算术模2^n定律，其中n是该类型机器表示的位数。例如，如果 char对象占用8位，那么unsigned char变量的取值范围为0~255，而signed char变量的取值范围则为- 128~127（在采用补码的机器上）。普通char对象是有符号的还是无符号的则取决于具体机器，但可打印字符总是正的。

long double类型用于指定高精度的浮点数。如同整数一样，浮点对象的大小也是由实现定义的，float、double与long double类型的对象可以具有同样大小，也可以表示两种或三种不同的大小。

在标准头文件<limits.h>与<float.h>中包含了有关所有这些类型的符号常量以及机器与编译程序的其他性质。这些内容将在附录 B中讨论。

练习2-1　编写一个程序来确定 signed及unsigned的char、short、int与long变量的取值范围，可以通过打印标准头文件中的相应值来完成，也可以直接计算来做。后一种方法较困难一些，因为要确定各种浮点类型的取值范围。

2.3　常量

诸如1234一类的整数常量是int常量。long常量要以字母 l或L结尾，如123456789L。一个整数常量如果大到在int类型中放不下，那么也被当做 long常量处理。无符号常量以字母 u或U结尾，后缀ul或UL用于表示unsigned long常量。

浮点常量中必须包含一个小数点（如 123.4）或指数（如1e-2）或两者都包含，在没有后缀时类型为double。后缀f与F用于指定float常量，而后缀l或L则用于指定long double常量。

整数值除了用十进制表示外，还可以用八进制或十六进制表示。如果一个整数常量的第一个数字为0，那么这个数就是八进制数；如果第一个数字为 0x或0X，那么这个数就是十六进制数。例如，十进制数31可以写成八进制数037，也可以写成十六进制数 0x1f或0X1F。在八进制与十六进制常量中也可以带有后缀 l或L（long，表示长八进制或十六进制常量）以及后缀 u或U（unsigned，表示无符号八进制或十六进制常量），例如，0XFUL是一个unsigned long 常量（无符号长整数常量），其值等于十进制数15。

字符常量是一个整数，写成用单引号括住单个字符的的形式，如 'x'。字符常量的值是该字符在机器字符集中的数值。例如，在 ASCII字符集中，字符 '0' 的值为48，与数值0没有关系。如果用字符 '0' 来代替像48一类的依赖于字符集的数值，那么程序会因独立于特定的值而更易于阅

读。虽然字符常量一般用来与其他字符进行比较，但字符常量也可以像整数一样参与数值运算。

有些字符用字符常量表示，这种字符常量是诸如“ \n ”（换行符）的换码序列，换码序列看起来像两个字符，但只用来表示一个字符。此外，我们可以用

```
'\ooo'
```

来指定字节大小的位模式，*ooo*是1~3个八进制数字（0…7）。位模式还可以用

```
'\xhh'
```

来指定，*hh*是一个或多个十六进制数字（0…9，a…f，A…F）。因此，可以如下写：

```
#define  VTAB  '\013'     /* ASCII纵向制表符 */
#define  BELL  '\007'     /* ASCII响铃符 */
```

也可以用十六进制写

```
#define  VTAB  '\xb'     /* ASCII纵向制表符 */
#define  BELL  '\x7'     /* ASCII响铃符 */
```

下面是所有的换码序列：

\a	响铃符	\ \	反斜杠
\b	回退符	\?	问号
\f	换页符	\'	单引号
\n	换行符	\"	双引号
\r	回车符	*ooo*	八进制数
\t	横向制表符	*xhh*	十六进制数
\v	纵向制表符		

字符常量 '\0' 表示其值为0的字符，即空字符。我们用 '\0' 来代替0，以在某些表达式中强调字符的性质，但其数字值就是0。

常量表达式是其中只涉及到常量的表达式。这种表达式可以在编译时计算而不必推迟到运行时，因而可以用在常量可以出现的任何位置，例如：

```
#define  MAXLINE  1000
char  line[MAXLINE+1];
```

或：

```
#define  LEAP  1    /* 闰年 */
int days[31+28+LEAP+31+30+31+30+31+31+30+31+30+31];
```

字符串常量也叫字符串字面值，是用双引号括住的由 0个或多个字符组成的字符序列。例如：

```
"I am a string"
```

或：

```
""   /* 空字符串 */
```

双引号不是字符串的一部分，它只用于限定字符串。在字符常量中使用的换码序列同样也可以用在字符串中，在字符串中用 \" 表示双引号字符。编译时可以将多个字符串常量连接起来：

```
"hello," " world"
```

等价于

```
"hello, world"
```

这种表示方法可用于将比较长的字符串分成若干源文件行。

从技术角度看，字符串常量就是字符数组。在内部表示字符串时要用一个空字符 '\0' 来结尾，故用于存储字符串的物理存储单元数比括在双引号中的字符数多一个。这种表示方法意味着， C 语言对字符串的长度没有限制，但程序必须扫描完整个字符串才能决定这个字符串的长度。标准库函数strlen(s) 用于返回其字符串变元 s的长度（不包括末尾的 '\0'。下面是我们设计的一个版本：

```
    /* strlen：返回s的长度 */
    int strlen(char s[])
    {
    int i;

    i = 0;
    while  (s[i] != '\0')
        ++i;
    return i;
}
```

strlen等字符串函数均说明在标准头文件 <string.h>中。

请仔细区分字符常量与只包含一个字符的字符串的区别： 'x'与 "x"不相同。前者是一个整数，用于产生字母 x在机器字符集中的数值（内部表示值）。后者是一个只包含一个字符（即字母 x）与一个 '\0'的字符数组。

另外还有一种常量，叫做枚举常量。枚举是常量整数值的列表，如同下面一样：

```
enum boolean { NO, YES };
```

在enum说明中第一个枚举名的值为 0，第二个为 1，如此等等，除非指定了显式值。如果不是所有值都指定了，那么未指定名字的值依着最后一个指定值向后递增，如同下面两个说明中的第二个说明：

```
enum escapes { BELL = '\a', BACKSPACE = '\b', TAB = '\t',
               NEWLINE = '\n', VTAB = '\v', RETURN = '\r' };

enum months { JAN = 1, FEB, MAR, APR, MAY, JUN,
              JUL, AUG, SEP, OCT, NOV, DEC };
                /* FEB的值为2，MAR的值为3，等等。*/
```

不同的枚举中的名字必须各不相同，同一枚举中各个名字的值不要求不同。

枚举是使常量值与名字相关联的又一种方便的方法，其相对于 #define语句的优势是常量值可以由自己控制。虽然可以说明 enum类型的变量，但编译程序不必检查在这个变量中存储的值是否为该枚举的有效值。枚举变量仍然提供了做这种检查的机会，故其比 #define更具优势。此外，调试程序能以符号形式打印出枚举变量的值。

2.4 说明

除了某些可以通过上下文做的隐式说明外，所有变量都必须先说明后使用。说明中不仅要

指定类型，还要包含由一个或多个该类型的变量组成的变量表。例如：

```
int  lower, upper, step;
char  c, line [1000];
```

同一类型的变量可以以任何方式分散在多个说明中，上面两个说明也可以等价地写成如下五个说明：

```
int  lower;
int  upper;
int  step;
char  c;
char  line [1000];
```

后一种形式需要占用较多的空间，但这样不仅便于向各个说明中增添注解，也便于以后的修改。

变量在说明时可以同时初始化。如果所说明的变量名后跟一个等号与一个表达式，那么这个表达式被作为初始化符。例如：

```
char  esc = '\\';
int  i = 0;
int  limit =MAXLINE+1;
float eps = 1.0e-5;
```

如果所涉及的变量不是自动变量，那么只初始化一次，而且从概念上讲应该在程序开始执行之前进行，此时要求初始化符必须为常量表达式。显式初始化的自动变量每当进入其所在的函数或分程序时就进行一次初始化，其初始化符可以是任何表达式。外部变量与静态变量的缺省初值为0。未经显式初始化的自动变量的值为未定义值（即为垃圾）。

在变量说明中可以用const限定符限定，该限定符用于指定该变量的值不能改变。对于数组，const限定符使数组所有元素的值都不能改变：

```
const double e = 2.71828182845905;
const char msg [ ] = "warning:";
```

const说明也可用于数组变元，表明函数不能改变数组的值：

```
int  strlen(const char[]);
```

如果试图修改const限定的值，那么所产生的后果取决于具体实现。

2.5 算术运算符

二元算术运算符包括+、-、*、/以及取模运算符%。整数除法要截取掉结果的小数部分。表达式

```
x % y
```

的结果是x除以y的余数，当y能整除x时，x % y的结果为0。例如，如果某一年的年份能被4整除但不能被100整除，那么这一年就是闰年，此外，能被400整除的年份也是闰年。因此，有

```
if ( (year % 4 == 0 && year % 100 != 0) || year % 400 == 0 )
    printf( "%d is a leap year\n", year );
else
    printf("%d is not a leap year\n", year );
```

取模运算符%不能作用于float或double对象。在有负的运算分量时，整数除法截取的方向以及取模运算结果的符号于具体机器，在出现上溢或下溢时所要采取的动作也取决于具体机。

二元＋和－运算符的优先级相同，但它们的优先级比 *、/和%的优先级低，而后者又比一元＋和－运算符低。算术运算符采用从左至右的结合规则。

本章末尾的表2-1总结了所有运算符的优先级和结合律。

2.6 关系运算符与逻辑运算符

关系运算符有如下几个：

```
>   >=   <   <=
```

所有关系运算符具有相同的优先级。优先级正好比它们低一级的是相等运算符：

```
==   !=
```

关系运算符的优先级比算术运算符低。因而表达式

```
i < lim -1
```

等于

```
i < (lim - 1)
```

更有趣的是逻辑运算符 &&与||。由&&与||连接的表达式从左至右计算，并且一旦知道结果的真假值就立即停止计算。绝大多数 C程序利用了这些性质。例如，下面这个循环语句取自第１章的输入函数getline：

```
for  ( i = 0; i < lim -1 &&(c = getchar()) != '\n' && c != EOF; ++i )
     s[i] = c;
```

在读一个新字符之前必须先检查一下在数组 s中是否还有空间存放这个字符，因此首先必须测试是否i < lim -1。而且，如果这一测试失败（即i < lim -1不成立），那么就没有必要继续读下一字符。

类似地，如果在调用 getchar函数之前就对 c是否为EOF进行测试，那么也是令人遗憾的，因此，函数调用与赋值都必须在对 c中的字符进行测试之前完成。

&&运算符的优先级比 ||运算符的优先级高，但两者都比关系运算符和相等运算符的优先级低。从而，像

```
i < lim - 1 && (c = getchar()) != '\n' && c != EOF
```

之类的表达式就不需要另外加圆括号了。但是，由于 !=运算符的优先级高于赋值运算符 =的优先级，在子表达式

```
(c = getchar()) != '\n'
```

中，圆括号还是需要的，这样才能达到我们所希望的先把函数值赋给 c再与 '\n' 进行比较的效果。

按照定义，如果关系表达式与逻辑表达式的计算结果为真，那么它们的值为 1；如果为假，那么它们的值为0。

一元求反运算符 !用于将非0运算分量转换成 0，把0运算分量转换成 1。该运算符通常用在诸如

```
if ( !valid )
```

一类的构造中，一般不用

```
if ( valid == 0 )
```

来代替。要想笼统地说哪个更好比较难。诸如 !valid一类的构造读起来好听一点（“如果不是有效的”），但这种形式在比较复杂的情况下可能难于理解。

练习2-2 不使用&&或||运算符编写一个与上面的for循环语句等价的循环语句。

2.7 类型转换

当一个运算符的几个运算分量的类型不相同时，要根据一些规则把它们转换成某个共同的类型。一般而言，只能把“比较窄的”运算分量自动转换成“比较宽的”运算分量，这样才能不丢失信息，例如，在诸如

```
f + i
```

一类的表达式的计算中要把整数变量i的值自动转换成浮点类型。不允许使用没有意义的表达式，例如，不允许把float表达式用作下标。可能丢失信息的表达式可能会招来警告信息，如把较长整数类型的值赋给较短整数类型的变量，把浮点类型赋给整数类型，等等，但不是非法表达式。

由于char类型就是小整数类型，在算术表达式中可以自由地使用 char类型的变量或常量。这就使得在某些字符转换中有了很大的灵活性。一个例子是用于将数字字符串转换成对应的数值的函数atoi：

```
/* atoi：将字符串s转换成整数 */
int atoi( char s[])
{
     int  i, n;

     n = 0;
     for ( i = 0; s[i] >= '0' && s[i] <= '9'; ++i )
         n = 10 * n + (s[i] - '0');
     return n;
}
```

正如第１章所述，表达式

```
s[i] - '0'
```

用于求s[i]中存储的字符所对应的数字值，因为 '0'、'1'、'2' 等的值形成不一个连续的递增序列。

将字符转换成整数的另一个例子是函数 lower，它把ASCII字符集中的字符映射成对应的小写字母。如果所要转换的字符不是大写字母，那么 lower函数返回原来的值。

```
/* lower：把字符c转换成小写字母；仅对ASCII字符集 */
int lower (int c)
{
        if (c >= 'A' && c<= 'Z' )
            return  c + 'a' - 'A';
    else
```

```
        return c;
}
```

这个函数是为ASCII字符集设计的。在ASCII字符集中，大写字母与对应的小写字母像数字值一样有固定的距离，并且每一个字母都是连续的——在A至Z之间只有字母。然而，后一个结论对于EBCDIC字符集不成立，故这一函数在EBCDIC字符集不只是转换了字母。

附录B中介绍的标准头文件 <ctype.h>定义了一组用于进行独立于字符集的测试和转换的函数。例如，tolower(c)函数用于在c为大写字母时将之转换成小写字母，故 tolower是上述lower函数的替代函数。同样条件，

```
c >= '0' && c<= '9'
```

可以用

```
isdigit(c)
```

代替。

我们从现在起要使用<ctype.h>中定义的函数。

在将字符转换成整数时有一点比较微妙。 C语言没有指定char类型变量是无符号量还是有符号量。当把一个char类型的值转换成int类型的值时，其结果是不是为负整数？结果视机器的不同而有所变化，反映了不同机器结构之间的区别。在某些机器上，如果字符的最左一位为 1，那么就被转换成负整数（称做“符号扩展”）。在另一些机器上，采取的是提升的方法，通过在最左边加上0把字符提升为整数，这样转换的结果总是正的。

C语言的定义保证了机器的标准打印字符集中的字符不会是负的，故在表达式中这些字符总是正的。但是，字符变量存储的位模式在某些机器上可能是负的，而在另一些机器上却是正的。为了保证程序的可移植性，如果要在 char变量中存储非字符数据，那么最好指定 signed或unsigned限定符。

关系表达式（如i > j）和由&&与||连接的逻辑表达式的值在其结果为真时为 1，在其结果为假时为0。因此，赋值语句

```
d = c >= '0' && c <= '9'
```

在c的值为数字时将d置为1，否则将d置为0。然而，诸如isdigit一类的函数在变元为真时返回的可能是任意非0值。在if、while、for等语句的测试部分，“真”的意思是“非0”，从这个意义上看，它们没有什么区别。

我们很希望能进行隐式算术类型转换。一般而言，如果诸如 + 或 * 等二元运算符的两个运算分量具有不同的类型，那么在进行运算之前先要把“低”的类型提升为“高”的类型。附录A.6节严格地给出了转换规则。然而如果没有无符号类型的运算分量，那么只要使用如下一组非正式的规则就够了：

如果某个运算分量的类型为long double，那么将另一个运算分量也转换成long double类型；
否则，如果某个运算分量的类型为double，那么将另一个运算分量也转换成double类型；
否则，如果某个运算分量的类型为float，那么将另一个运算分量也转换成float类型；
否则，将char与short类型的运算分量转换成int类型。
然后，如果某个运算分量的类型为long，那么将另一个运算分量也转换成long类型。

注意，在表达式中float类型的运算分量不自动转换成double类型，这与原来的定义不同。一般而言，数学函数（如定义在标准头文件 <math.h>中的函数）要使用双精度。使用 float类型的主要原因是为了在使用较大的数组时节省存储单元，有时也为了节省机器执行时间（双精度算术运算特别费时）。

当表达式中包含unsigned类型的运算分量时，转换规则要复杂一些。主要问题是，在有符号值与无符号值之间的比较运算取决于机器，因为它们取决于各个整数类型的大小。例如，假定int对象占16位，long对象占32位，那么，-1L < 1U，这是因为int类型的-1U被提升为signed long类型；但-1L > 1UL，这是因为-1L被提升为unsigned long类型，因此它是一个比较大的正数。

在进行赋值时也要进行类型转换，= 右边的值要转换成左边变量的类型，后者即赋值表达式结果的类型。

如前所述，不管是否要进行符号扩展，字符值都要转换成整数值。

当把较长的整数转换成较短的整数或字符时，要把超出的高位部分丢掉。于是，当程序

```
int  i;
char c;

i = c;
c = i;
```

执行后，c的值保持不变，无论是否要进行符号扩展。然而，如果把两个赋值语句的次序颠倒一下，那么执行后可能会丢失信息。

如果x是float类型且、i是int类型，那么

```
x = I
```

与

```
i = x
```

这两个赋值表达式在执行时都要引起类型转换 ,当把float类型转换成int类型时要把小数部分截取掉；当把float类型转换成int类型时，是四舍五入还是截取取决于具体实现。

由于函数调用的变元是表达式，当把变元传递给函数时也可能引起类型转换。在没有函数原型的情况下，char与short类型转换成int类型，float类型转换成double类型，这就是即使在函数是用char与float类型的变元表达式调用时仍把参数说明成 int与double类型的原因。

最后，在任何表达式中都可以进行显式类型转换（即所谓的“强制转换”），这时要使用一个叫做强制转换的一元运算符。在如下构造中，表达式被按上述转换规则转换成由类型名所指名的类型：

（类型名）表达式

强制转换的精确含义是，表达式首先被赋给类型名指定类型的某个变量，然后再将其用在整个构造所在的位置。例如，库函数 sqrt需要一个double类型的变元，但如果在其他地方作了不适当的处理，那么就会产生无意义的结果（ sqrt是在<math.h>中说明的一个函数）。因而，如果n是整数，那么可以用

```
sqrt ( (double) n)
```

使得在把n传递给sqrt函数之前先把n的值转换成double类型。注意，强制转换只是以指名的类型产生n的值，n本身的值没有改变。强制转换运算符与其他一元运算符具有相同的优先级，如同本章末尾的表中所总结的那样。

如果变元是通过函数原型说明的，那么在通常情况下，当该函数被调用时，系统对变元自动进行强制转换。于是，对于sqrt的函数原型

```
double sqrt(double);
```

调用

```
root = sqrt(2);
```

不需要强制转换运算符就自动将把整数2强制转换成double类型的值2.0。

在标准库中包含了一个用于实现伪随机数发生器的函数rand与一个用于初始化种子的函数srand。在前一个函数中使用了强制转换：

```
unsigned long int next = 1;

/* rand：返回取值在0~32767之间的伪随机数 */
int rand(void)
{
    next = next *1103515245 +12345;
    return (unsigned int)(next/65536) % 32768;
}

/* srand：为rand()函数设置种子 */
void srand(unsigned int seed)
{
    next = seed;
}
```

练习2-3 编写函数htoi(s)，把由十六进制数字组成的字符串（前面可能包含0x或0X）转换成等价的整数值。字符串中允许的数字为：0~9，a~f、以及A~F。

2.8 加一与减一运算符

C语言为变量增加与减少提供了两个奇特的运算符。加一运算符++用于使其运算分量加1，减一运算符--用于使其运算分量减1。我们常常用++运算符来使变量的值加1，如在下述语句中一样：

```
if  (c == '\n')
    ++nl;
```

++与--这两个运算符奇特的方面在于，它们既可以用作前缀运算符（用在变量前面，如++n），也可以用作后缀运算符（用在变量后面，如n++）。在这两种情况下，效果都是使n加1。但是，它们之间仍存在一点区别，表达式

```
++n
```

在n的值被使用之前先使n加1，而表达式

```
n++
```

则是在n的值被使用之后再使n加1。这意味着，在该值被使用的上下文中， ++n和n++的效果是不同的。如果n的值是５，那么

```
x = n++;
```

将x的值置为5，而

```
x = ++n;
```

则将x的值置为６。在这两个语句执行完后， n的值都是６。加一与减一运算符只能作用于变量，诸如

```
(i + j)++
```

一类的表达式是非法的。

在除了加1的运算效果，不需要任何具体值的地方，如表达式

```
if  (c == '\n')
    nl++;
```

中，++作为前缀与后缀效果是一样的。在有些情况下需要特别指定。例如，考虑下面的函数squeeze(s,c)，它用于从字符串s中把所有出现的字符c都删除掉：

```
/* squeeze：从s中删除掉c  */

void squeeze(char s[], int c)
{
    int  i, j;
    for (i = j = 0; s[i] != '\0'; i++)
        if (s[i] != c)
            s[j++] = s[i];
    s[j] = '\0';
}
```

每当出现一个不等于c的字符时，就把它拷贝到j的当前值所指向的位置，并将j的值加1，以准备处理下一个字符。其中的if语句完全等价于

```
if (s[i] != c) {
    s[j] = s[i];
    j++;
}
```

具有类似构造的另一个例子是第１章的getline函数。我们可以将这个函数中的if语句

```
if (c == '\n') {
    s[i] = c;
    ++i;
}
```

用更为精致的if语句

```
if (c == '\n')
    s[i++] = c;
```

代替。

作为第三个例子，再看一下标准函数 strcat(s,t)，它用于把字符串 t连接到字符串 s的后面。strcat函数假定在s中有足够的空间来保存这两个字符串连接的结果。下面所编写的这个函数没有返回任何值（在标准库中，这个函数要返回一个指向新字符串的指针）：

```
/* strcat：把字符串t连接到字符串s的后面；s必须有足够大的空间  */
void strcat(char s[], char t[])
{
     int i, j;

     i = j = 0;
     while (s[i] != '\0')    /*  找到s的末尾  */
         i++;
     while ( (s[i++] = t[j++]) != '\0')    /*  拷贝t  */
         ;
}
```

在将t中的字符逐个拷贝到s后面时，用后缀运算符 ++ 作用于i与j，以保证在循环过程中i与j均指向下一个位置。

练习2-4　重写squeeze(s1,s2)函数，把字符串s1中与字符串s2中字符匹配的各个字符都删除掉。

练习2-5　编写函数any(s1,s2)，它把字符串s2中任一字符在字符串s1中的第一次出现的位置作为结果返回。如果s1中没有包含s2中的字符，那么返回-1。（标准库函数strpbrk具有同样的功能，但它返回的是指向该位置的指针。）

2.9 按位运算符

C语言提供了六个用于位操作的运算符，这些运算符只能作用于整数分量，即只能作用于有符号或无符号的char、short、int与long类型：

```
&      按位与（AND）
|      按位或（OR）
^      按位异或（XOR）
<<     左移
>>     右移
~      求反码（一元运算符）
```

按位与运算符&经常用于屏蔽某些位，例如：

```
n = n & 0177;
```

用于将n除7个低位外的各位置成0。

按位或运算符 | 用于打开某些位，例如：

```
x = x | SET_ON;
```

用于将x中与SET_ON中为1的位对应的那些位也置为1。

按位异或运算符^用于在其两个运算分量的对应位不相同时置该位为 1，否则，置该位为0。

我们必须将按位运算符 &和 | 同逻辑运算符&&和 || 区分开来，后者用于从左至右求表达式的真值。例如，如果x的值为1，y的值为2，那么，x & y的结果是0，而x && y的值则为1。

移位运算符<<与>>分别用于将左运算分量左移与右移由右运算分量所指定的位数（右运算分量的值必须是正的）。于是，表达式x << 2用于将x的值左移2位，右边空出的2位用0填空，这个表达式的结果等于左运算分量乘以 4。当右移无符号量时，左边空出的部分用 0填空；当右移有符号的量时，在某些机器上对左边空出的部分用符号位填空（即“算术移位”），而在另一些机器上对左边空出的部分则用0填空（即“逻辑移位”）。

一元~ 运算符用于求整数的反码，即它分别将运算分量各位上的 1转换成0，0转换成1。例如：

```
x = x & ~077
```

用于将x的最后六位置为0。注意，表达式x &~077是独立于字长的，它要比诸如 x & 0177700一类的表达式好，后者假定x是16位的量。这种可移植的形式并没有增加额外开销，因为 ~ 077是常量表达式，可以在编译时求值。

为了对某些按位运算符做进一步说明，考虑函数 getbits(x, p, n)，它用于返回x从p位置开始的（右对齐的） n位的值。假定第 0位是最右边的一位， n与p都是符合情理的正值。例如，getbits(x, 4, 3)返回右对齐的第4、3、2共三位：

```
/* getbits：取从第p位开始的n位 */
unsigned getbits(unsigned x, int p, int n)
{
    return  (x >> (p+1-n) ) & ~(~ 0 << n);
}
```

其中的表达式 x >> (p+1-n) 将所希望的位段移到字的右边。 ~ 0 将所有位都置为1，~ 0 << n将(~) 0左移n位，将最右边的n位用0填空。再对这个表达式求反，将最右边 n位置为1，其余各位置为0。

练习2-6 编写一个函数setbits(x, p, n, y)，返回对x做如下处理得到的值：x从第p位开始的n位被置为y的最右边n位的值，其余各位保持不变。

练习2-7 编写一个函数invert(x, p, n)，返回对x做如下处理得到的值：x从第p位开始的n位被求反（即，1变成0，0变成1），其余各位保持不变。

练习2-8 编写一个函数rightrot (x, n)，返回将x向右循环移动n位所得到的值。

2.10 赋值运算符与赋值表达式

在一个赋值表达式⊖中，如果赋值运算符左边的变量在右边紧接着又要重复一次，如：

```
i = i +2
```

那么可以将这种表达式改写成更精简的形式：

```
i += 2
```

⊖ 作者这里所说的赋值运算符实际上专指复合赋值运算符（ +=、-=、*=、/=、%=、>>=、<<=、&=、^=与!=），没有包含简单赋值运算符（ =）。严格来讲，赋值运算符应包含简单赋值运算符与复合赋值运算符两类。——译者注

其中的运算符 += 叫做赋值运算符。

大多数二元运算符（即有左右两个运算分量的运算符）都有一个对应的赋值运算符 *op*=，*op* 是下面这些运算符中的一个：

```
+  -  *  /  %  <<  >>  &  ^  |
```

且表达式

表达式 *op* = 表达式$_2$

等价于

表达式$_1$ = (表达式$_1$) *op* (表达式$_2$)

区别在于，在前一种形式中表达式$_1$只计算一次。注意，表达式$_1$与表达式$_2$两边的圆括号，它们是不可少的，如：

```
x *= y + 1
```

的意思是：

```
x = x * (y + 1)
```

而不是：

```
x = x * y + 1
```

例如，下面的函数bitcount用于统计其整数变元中值为1的位的个数：

```
/* bitcount：统计x中值为1的位数 */
int bitcount (unsigned x )
{
     int  b;

     for  ( b = 0; x != 0; x >>= 1)
          if  ( x & 01 )
          b++;
   return b;
}
```

将x说明为无符号整数是为了保证：当将 x右移时，不管该函数运行于什么机器上，左边空出的各位能用0（而不是符号位）填满。

除了简明外，这类赋值运算符还有一个其表示方式与人们的思维习惯比较接近的优点。我们通常会说“把2加到i上”或“i加上2”，而不会说“取i，加上2，再把结果放回到i中”，因此，表达式i += 2比i = i + 2好。此外，对于诸如

```
yyval [ yypv [ p3 + p4 ] + yypv [ p1 + p2 ] ] += 2
```

等更复杂的表达式，这种赋值运算符使程序代码更易于理解，读者不需要煞费苦心地去检查两个长表达式是否完全一样，也无需为两者为什么不一样而感到疑惑不解。而且，这种赋值运算符还有助于编译程序产生高效的目标代码。

我们已经看到，赋值语句⊖有一个值，而且可以用在表达式中。最常见的例子是：

⊖ ANSI C中没有使用赋值语句这一术语，这里似乎应叫做赋值表达式。而且语句以分号结束，作为语句不能出现在表达式中。 —— 译者注

```
while  ( ( c = getchar( ) ) != EOF)
    …
```

其他赋值运算符（即复合赋值运算符+=、-=等）也可以用在表达式中，尽管这种用法比较少。

在所有这类表达式中，赋值表达式的类型就是左运算分量的类型，值也是在赋值后左运算分量的值。

练习2-9 在求反码时，表达式x &= (x - 1)用于把x最右边的值为1的位删除掉。请解释一下这样做的道理。用这一方法重写bitcount函数，使之执行得更快一点。

2.11 条件表达式

语句

```
if  (a > b)
    z = a;
else
    z = b;
```

用于求a与b中的最大值并将之放到z中。作为另一种方法，可以用条件表达式（使用三元运算符?:）来写这段程序及类似的代码段。在表达式

表达式$_1$? 表达式$_2$: 表达式$_3$

中，首先计算表达式$_1$，如果其值不等于0（即为真），则计算表达式$_2$的值，并以该值作为本条件表达式的值；否则计算表达式$_3$的值，并以该值作为本条件表达式的值。在表达式$_2$与表达式$_3$中，只有一个会被计算到。因此，以上语句可以改写成：

```
z = (a > b) ? a : b;       /* z = max(a, b) */
```

应该注意到，条件表达式就是一种表达式，它可以用在其他表达式能用的所有地方。如果表达式$_2$与表达式$_3$具有不同的类型，那么结果的类型由本章前面讨论的转换规则决定。例如，如果f为float类型，n为int类型，那么表达式

```
(n > 0) ? f : n
```

的类型为float，无论n是不是正的。

条件表达式中用于括住第一个表达式的圆括号并不是必需的，这是因为条件运算符 ?: 的优先级非常低，仅高于赋值运算符。但我们还是建议使用圆括号，因为这可以使表达式的条件部分更易于阅读。

条件表达式可用于编写简洁的代码。例如，下面的循环语句用于打印一个数组的 n个元素，每行打印10个元素，每一列之间用一个空格隔开，每行用一个换行符结束（包括最后一行）：

```
for (i = 0; i , n; i++)
    printf("%6d%c", a[i], (i % 10 == 9 || i == n - 1) ? '\n' : ' ');
```

在每10个元素之后以及在第n个元素之后都要打印一个换行符，所有其他元素后都要跟一个空格，这看起来有点麻烦，但要比相应的if-else结构紧凑。下面是另一个使用条件运算符的好例子：

```
printf("you have %d item%s.\n", n, n == 1 ? "" : "s");
```

练习2-10 重写用于将大写字母转换成小写字母的函数lower，用条件表达式替代其中的if-

else结构。

2.12 运算符优先级与表达式求值次序

表2-1总结了所有运算符的优先级与结合律规则，包括尚未讨论的一些规则。同一行的各个运算符具有相同的优先级，纵向看越往下优先级越低。例如，*、/与%三者具有相同的优先级，它们的优先级都比二元+与-运算符高。运算符( )指函数调用。运算符->与.用于访问结构成员，第6章将讨论这两个运算符以及sizeof（对象大小）运算符。第5章将讨论运算符*（用指针间接访问）与&（对象的地址），第3章将讨论逗号（,）运算符。

表2-1 运算符优先级与结合律

运 算 符	结 合 律
( ) [ ] -> .	从左至右
! ~ ++ -- + - * & (类型) sizeof	从右至左
* / %	从左至右
+ -	从左至右
<< >>	从左至右
< <= > >=	从左至右
== !=	从左至右
&	从左至右
^	从左至右
|	从左至右
&&	从左至右
||	从左至右
?:	从右至左
= += -= *= /= %= &= ^= |= <<= >>=	从右至左
,	从左至右

注：一元+、-与*运算符的优先级比相应二元运算符高。

注意，按位运算符 &、^ 与 | 的优先级比相等运算符 == 与 != 低。这意味着，在诸如

```
if  ( (x & MASK) == 0)
   …
```

中，位测试表达式必须用圆括号括起来，才能得到正确的结果。

如同大多数语言一样，C语言没有指定同一运算符的几个运算分量的计算次序（&&、||、?:与，除外）。例如，在诸如

```
x = f( ) + g( );
```

一类的语句中，f( )可以在g( )之前计算，也可以在g( )之后计算。因此，如果函数f或g中改变了另一个函数所要使用的变量的值，那么x的结果值可能依赖于这两个函数的计算次序。为了保证特定的计算次序，可以把中间结果保存到临时变量中。

同样，在函数调用中各个变元的求值次序也是未指定的。因而，函数调用语句

```
printf("%d %d\n", ++n, power(2,n) );    /* 错 */
```

对不同的编译程序可能会产生不同的结果（视 n加一运算是在power调用之前还是之后而定）。为了解决这一问题，可把该语句改写成

```
++n;
    printf("%d %d\n", n, power(2,n) );
```

函数调用、嵌套的赋值语句、加一与减一运算符都有可能引起“副作用”——作为表达式求值的副产品，改变了某些变量的值。在涉及到副作用的表达式中，对作为表达式一部分的本来的求值次序存在着微妙的依赖关系。下面的表达式语句是这种使人讨厌的情况的一个典型例子：

```
a[i] = i++;
```

问题是，数组下标的值i是旧值还是新值。编译程序对之可以有不同的解释，并视不同的解释产生不同的结果。C语言标准故意留下了许多诸如此类的问题未作具体规定。何时处理表达式中的副作用（对变量赋值）是各个编译程序的事情，因为最好的求值次序取决于机器结构。（标准明确规定了所有变元的副作用都必须在该函数被调用之前生效，但这对上面对 printf函数的调用没有什么好处。）

从风格角度看，在用任何语言编写程序时，编写依赖于求值次序的代码不是一种好的程序设计习惯。很自然地，我们需要知道哪些事情需要避免，但如果不知道它们在各种机器上是如何执行的，那么不要试图去利用特定的实现。

第3章 控 制 流

一个语言的控制流语句用于指定各个计算执行的次序。在前面的例子中我们已经见到了一些最常用的控制流结构。本章将全面讨论控制流语句，更精确、更全面地对它们进行介绍。

3.1 语句与分程序

在诸如x =0、i++或printf (...)之类的表达式之后加上一个分号（;），就使它们变成了语句[⊖]：

```
x = 0;
i++;
printf(……);
```

在C语言中，分号是语句终结符，而不是像Pascal等语言那样把分号用做语句之间的分隔符。

可以用一对花括号{与}把一组说明和语句括在一起构成一个复合语句（也叫分程序），复合语句在语法上等价于单个语句，即可以用在单个语句可以出现的所有地方。一个明显的例子是在函数说明中用花括号括住的语句，其他的例子有在if、else、while与for之后用花括号括住的多个语句。（在任何分程序中都可以说明变量，第4章将对此进行讨论。）在用于终止分程序的右花括号之后不能有分号。

3.2 if-else语句

if-else语句用于表示判定。其语法形式如下：

```
if  (表达式)
    语句1
else
    语句2
```

其中else部分是任选的。在if语句执行时，首先计算表达式的值，如果其值为真（即，如果表达式的值非0），那么就执行语句$_1$；如果其值为假（即，如果表达式的值为0），并且包含else部分，那么就执行语句$_2$。

由于if语句只是测试表达式的数值，故表达式可以采用比较简捷的形式。最明显的例子是用

```
if  (表达式)
```

代替

```
if  (表达式 != 0 )
```

有时这样既自然又清楚，但有时又显得比较隐秘。

由于if-else语句的else部分是任选的，当在嵌套的if语句序列中缺省某个else部分时会引起歧

⊖ 在表达式后加上分号构成的语句叫做表达式语句。 —— 译者注

义。这个问题可以通过使每一个else与最近的还无else匹配的if匹配。例如，在如下语句中：

```
if  ( n > 0 )
    if  ( a > b )
        z = a;
    else
        z = b;
```

else部分与嵌套在里面的if匹配，正如缩入结构所表示的那样。如果这不是我们所希望的，那么可以用花括号来使该else部分与所希望的if强制结合：

```
if  ( n > 0 )  {
    if  ( a > b )
        z = a;
}
else
    z = b;
```

歧义性在有些情况下特别有害，例如，在如下程序段中：

```
if  ( n >= 0 )
    for ( i = 0; i < n; i++ )
        if  ( s[i] > 0)  {
            printf ( "…" );
            return i;
        }
else              /*  错  */
    printf ( "error -- n is negative\n" );
```

其中的缩入结构明确地给出了我们所希望的结果，但编译程序无法得到这一信息，它会使else部分与嵌套在里面的if匹配。这种错误很难发现，因此我们建议在if语句嵌套的情况下尽可能使用花括号。

顺便请读者注意，在语句

```
if  ( a > b )
    z = a;
else
    z = b;
```

中，在z = a后有一个分号。这是因为，从语法上讲，跟在if后面的语句总是以一个分号终结，诸如z = a之类的表达式语句也不例外。

3.3 else-if 语句

在C程序经常使用如下结构：

```
if  ( 表达式 )
    语句
else if  ( 表达式 )
    语句
else if  ( 表达式 )
    语句
```

```
else if  ( 表达式 )
    语句
else
    语句
```

由于这种结构经常要用到，值得单独对之进行简要讨论。这种嵌套的 if语句构成的序列是编写多路判定的最一般的方法。各个表达式依次求值，一旦某个表达式为真，那么就执行与之相关的语句，从而终止整个语句序列的执行。每一个语句可以是单个语句，也可以是用花括号括住的一组语句。

最后一个else部分用于处理“上述条件均不成立”的情况或缺省情况，此时，上面的各个条件均不满足。有时对缺省情况不需要采取明显的动作，在这种情况下，可以把该结构末尾的

```
else
    语句
```

省略掉，也可以用它来检查错误，捕获“不可能”的条件。

可以通过一个二分查找函数来说明三路判定的用法。这个函数用于判定在数组 v中是否有某个特定的值x。数组v的元素必须以升序排列。如果在 v中包含x，那么该函数返回 x在v中的位置（介于0~n-1之间的一个整数）；否则，该函数返回- 1。

在二分查找时，首先将输入值 x与数组v的中间元素进行比较。如果 x小于中间元素的值，那么在该数组的前半部查找；否则，在该数组的后半部查找。在这两种情况下，下一步都是将 x与所选一半的中间元素进行比较。这一二分过程一直进行下去，直到找到指定的值，或查找范围为空。

```
/*  binsearch：在v[0]<=v[1]<=v[2]<=……<=v[n-1]中查找x  */
int binsearch ( int x, int v[ ], int n )
{
    int  low, high, mid;

    low = 0;
    high = n - 1;
    while  ( low <= high )  {
        mid = ( low + high ) / 2;
        if  ( x < v[mid] )
            high = mid - 1;
    else if  ( x > v[mid] )
        low = mid + 1;
    else       /*  找到了匹配的值  */
        return  mid;
    }
    return  -1;     /* 没有查到 */
}
```

这个函数的基本判定是，在每一步 x是小于、大于还是等于中间元素 v[mid]，这自然就用到了else-if结构。

练习3-1　在上面有关二分查找的例子中，在 while循环语句内共作了两次测试，其实只要一

次就够了（以把更多的测试放在外面为代价）。重写这个函数，使得在循环内部只进行一次测试，并比较两者运行时间的区别。

3.4 switch语句

switch语句是一种多路判定语句，它根据表达式是否与若干常量整数值中的某一个匹配来相应地执行有关的分支动作。

```
switch  ( 表达式 )  {
case  常量表达式:  语句序列
case  常量表达式:  语句序列
default:  语句序列
}
```

每一种情形都由一个或多个整数值常量或常量表达式标记。如果某一种情形与表达式的值匹配，那么就从这个情形开始执行。各个情形中的表达式必须各不相同。如果没有一个情形能满足，那么执行标记为default的情形。default情形是任选的。如果没有default情形并且没有一个情形与表达式的值匹配，那么该switch语句不执行任何动作。各个情形及default情形的出现次序是任意的。

第1章曾用if...else if...else结构编写过一个程序来统计各个数字、空白字符及所有字符出现的次数。下面是用switch语句改写的程序：

```
#include  <stdio.h>

main ( )    /* 统计数字、空白及其他字符 */
{
      int  c, i, nwhite, nother, ndigit[10];

      nwhite = nother = 0;
      for  ( i = 0; i < 10, i++ )
          ndigit[i] = 0;
      while  ( ( c = getchar ( ) ) != EOF )  {
          switch  ( c ) {
          case '0': case '1': case '2': case '3': case '4':
          case '5': case '6': case '7': case '8': case '9':
              ndigit[c - '0']++;
              break;
          case ' ':
          case '\n':
          case '\t':
              nwhite++;
              break;
          default:
              nother++;
              break;
          }
      }
```

```
        printf ( "digits = " );
        for  ( i = 0; i < 10, i++ )
            printf ( "  %d", ndigit[i] );
    printf ( ",  white space = %d,  other = %d\n",  nwhite,  nother );
        return  0;
    }
```

break语句用于从switch语句中退出。由于在switch语句中case情形的作用就像标号一样，在某个case情形之后的代码执行完后，就进入下一个 case情形执行，除非显式控制转出。转出switch语句最常用的方法是使用break语句与return语句。break语句还可用于从while、for与do循环语句中立即强制性退出，对于这一点，稍后将做进一步讨论。

对于依次执行各种情形这种做法毁誉参半，好的一方面，它可以把若干个情形组合在一起完成某个任务，如上例中对数字的处理。但是，为了防止直接进入下一个情形执行，它要求在每一个情形后以一个break语句结尾。从一个情形直接进入下一个情形执行这种做法不是一种健全的做法，在程序修改时很容易出现错误。除了将多个标号用于表示同一计算的情况外，应尽量少从从一个情形直接进入下一个情形执行并在不得不使用时加上适当的注解。

作为一种好的风格，可以在switch语句最后一个情形（即default情形）后加上一个break语句，虽然这样做在逻辑上没有必要。但当以后需要在该 switch语句后再添加一种情形时，这种防范型程序设计会使我们少犯错误。

练习3-2　编写函数escape (s, t)，将字符串t拷贝到字符串s中，并在拷贝过程中将诸如换行符与制表符等等字符转换成诸如 \n与\t等换码序列。使用 switch语句。再编写一个具有相反功能的函数，在拷贝过程中将换码序列转换成实际字符。

3.5　while与for循环语句

我们在前面已经遇到过while与for循环语句。在while循环语句

```
while  ( 表达式 )
    语句
```

执行中，首先求表达式的值。如果其值不等于零，那么就执行语句并再次求该表达式的值。这一周期性过程一直进行下去，直到该表达式的值变为假，此时从语句的下一个语句接着执行。

for循环语句

```
for  ( 表达式1; 表达式2; 表达式3 )
    语句
```

等价于

```
表达式1;
while  ( 表达式2 )  {
    语句
    表达式3;
}
```

但包含contine语句时的行为除外，该语句将在3.7节中介绍。

从语法上看，for循环语句的三个组成部分都是表达式。最常见的情况是，表达式$_1$与表达式$_3$

是赋值表达式或函数调用，表达式$_2$是关系表达式。这三个表达式中任何一个都可以省略，但分号必须保留。如果表达式1与表达式$_3$被省略了，那么它退化成了while循环语句。如果用于测试的表达式$_2$不存在，那么就认为表达式$_2$的值永远是真的，从而，for循环语句

```
for  ( ; ; )  {
    …
}
```

就是一个“无限”循环语句，这种语句要用其他手段（如break语句或return语句）才能终止执行。

在这两种循环语句中到底选用while语句还是for语句主要取决于程序人员的个人爱好。例如，在如下语句中:

```
while  ( ( c = getchar ( ) ) == ' ' || c == '\n' || c == '\t' )
    ;       /* 跳过空白字符 */
```

不包含初始化或重新初始化部分，所以使用while循环语句最为自然。

如果要做简单地初始化与增量处理，那么最好还是使用 for语句，因为它可以使循环控制的语句更密切，而且它把控制循环的信息放在循环语句的顶部，易于程序理解。这在如下语句中表现得更为明显：

```
for  ( i = 0;  i < n;  i++ )
    …
```

这是C语言在处理一个数组的前n个元素时的一种习惯性用法，类似于 FORTRAN语言的DO循环语句与Pascal语言的for循环语句。但是，这种类比不够恰当，因为 C语言循环语句的位标值和终值在循环语句体内可以改变，在循环因某种原因终止时位标变量 i的值仍然保留。由于for语句的各个组成部分可以是任何表达式，故 for语句并不限于以算术值用于循环控制。然而，强制性地把一些无关的计算放到for语句的初始化或增量部分是一种很坏的程序设计风格，它们最好用做循环控制的操作。

作为一个更大的例子，再次考虑用于将字符串转换成对应数值的函数 atoi。下面这个版本要比第2章介绍的那个版本更为通用一些，它可以处理任何前导空白字符与加减号。（第4章将介绍另一个类似的函数atof，它用于对浮点数作同样的转换。）

程序的结构反映了输入的形式：

```
跳过可能有的空白字符
取可能有的符号
取整数部分并转换它
```

每一步都处理其输入，并给下一步留下一个清楚的状态。整个处理过程持续到不是数的一部分的第一个字符为止。

```
#include  <ctype.h>

/* atoi：将s转换成整数；第2版*/
int atoi ( char  s[ ])
{
   int  i, n , sign;

for ( i = 0;  isspace ( s[i] );  i++ )   /* 跳过空白字符 */
```

```
        ;
    sign =  ( s[i] == '-' ) ? -1 : 1;
    if  (s[i] == '+' || s[i] == '-' )    /* 跳过符号 */
       i++;
    for  ( n = 0;  isdigit ( s[i] );  i++)
        n = 10 * n + (s[i] - '0' );
    return  sign * n;
}
```

标准库中提供了一个更精巧的函数strtol，它用于把字符串转换成长整数，参见附录B.5节。

当使用嵌套循环语句时，把循环控制集中到一起的优点更为明显。下面的函数用于实现对整数数组进行排序的Shell排序法。这个排序算法是由D. L. Shell于1959年发明的，其基本思想是，先对隔得比较远的元素进行比较，而不是像简单交换排序算法中那样比较相邻的元素。这样可以快速地减少大量的无序情况，以后就可以少做些工作。各个被比较的元素之间的距离在逐步减少，一直减少到1，此时排序变成了相邻元素的互换。

```
/* shellsort：以递增顺序对v[0]、v[1]、……、v[n-1] 进行排序*/
void shellsort ( int v[ ], int n )
{
   int  gap, i, j, temp;

   for  ( gap = n/2;  gap > 0;  gap /= 2 )
        for  ( i = gap;  i < n;  i++ )
               for  ( j = i-gap;  j >= 0 && v[j] > v[j+gap];  j -= gap )  {
                      temp = v[j];
                      v[j] = v[j+gap];
                      v[j+gap] = temp;
        }
}
```

这个函数中包含三个嵌套的for循环语句。最外层的for语句用于控制两个被比较元素之间的距离，从n/2开始对折，一直到0。中间的for语句用于控制每一个元素。最内层的for语句用于比较各对相距gap个位置的元素，并在这两个元素的大小位置颠倒时把它们互换过来。由于gap的值最终要减到1，所有元素最终都会在正确的排序位置上。注意即使在非算术值的情况下，for语句的通用性也使得外层循环能够适应。

C语言还有一个运算符，叫做逗号运算符“ , ”，在for循环语句中经常要使用到它。由逗号分隔的各个表达式从左至右进行求值，结果的类型和值是右运算分量的类型和值。因此，在for循环语句中，可以把多个表达式放在不同的部分，例如，可以同时处理两个位标（控制变量）。这可以通过函数reverse (s)来说明，该函数用于把字符串s中各个字符的位置颠倒一下。

```
#include  <string.h>

/* reverse：颠倒字符串s中各个字符的位置 */
void reverse ( char s[ ] )
{
   int  c, i, j;
```

```
    for  ( i = 0, j =strlen(s) - 1;  i < j;  i++, j-- )  {
         c = s[i];
         s[i] = s[j];
         s[j] = c;
    }
}
```

用于分隔函数变元、说明中的变量等的逗号不是逗号运算符，对逗号运算符也不保证要从左至右求值。

应谨慎使用逗号运算符。逗号运算符最适合用于描述彼此密切相关的构造，如上面 reverse函数内的for语句中的逗号运算符以及需要在单个表达式中表示多步计算的宏。逗号表达式也适合用在renverse函数的元素交换过程中，该交换过程被当做单步操作。

```
for  ( i = 0, j =strlen(s) - 1;  i < j;  i++, j-- )
c = s[i],  s[i] = s[j],  s[j] = c;
```

练习3-3 编写函数expand(s1, s2)，将字符串s1中诸如a-z一类的速记等号在字符串s2中扩展成等价的完整列表abc……xyz。允许处理大小写字母和数字，并可以处理诸如 a-b-c与a-z0-9与-a-z等情况。正确安排好前导与尾随的“ - ”。

3.6 do-while循环语句

正如第1章所述，while与for这两个循环语句在循环体执行前对终止条件进行测试。与之相对应的，C语言中的第三种循环语句 —— do-while循环语句 —— 则是在循环体执行完后再测试终止条件，循环体至少要执行一次。

do-while循环语句的语法是：

```
do
    语句
while  ( 表达式 )
```

在do-while循环语句执行时，先要执行语句，然后再求表达式的值。如果表达式的值为真，那么就再次执行语句，如此等等。当表达式的值变成假的的时候，就终止循环的执行。除了条件测试的语义外，do-while循环语句与Pascal语言的repeat-until语句等价。

经验表明，使用do-while语句的场合要比使用while语句和for语句的场合少得多。然而，do-while循环语句有时还是很有价值的，如下面的函数 itoa。itoa函数是atoi函数的逆函数，用于把数字转换成字符串。这一工作要比想象的复杂，因为如果以产生数字的方法来产生字符串，所产生的字符串的次序正好颠倒了。故先生成颠倒的字符串，然后再把它颠倒过来。

```
/* itoa：将数字n转换成字符存到s中 */
void itoa ( int n,  char s[ ] );
{
   int  i, sign;

   if  ( ( sign = n ) < 0 )      /* 记录符号 */
       n = -n;                   /* 使n成为正数 */
   i = 0;
```

```
    do  {                           /* 以反序生成数字 */
        s[i++] = n % 10 + '0';      /* 取下一个数字 */
    } while ( (n /= 10) > 0);       /* 删除该数字 */
    if  (sign < 0)
        s[i++] = '-';
    s[i] = '\0';
    reverse(s);
}
```

因为即使n为0也要至少把一个字符放到数组 s中，所以在这里有必要使用 do-while语句，至少使用do-while语句要方便一些。我们也用花括号来括住作为 do-while语句体的单个语句，即使没有必要的这样做，但这样可以使那些比较轻率的读者在使用 while语句时少犯些错误。

练习3-4 在数的反码表示中，上述 itoa函数不能处理最大的负数，即 n为$-(2^{字长-1})$时的情况。解释其原因。对该函数进行修改，使之不管在什么机器上运行都能打印出正确的值。

练习3-5 编写函数itob(n, s, b)，用于把整数n转换成以b为基的字符串并存到字符串 c中。特别地，itob(n, s, 16)用于把n格式化成十六进制整数字符串并存在 s中。

练习3-6 修改itoa函数使之改为接收三个变元。第三个变元是最小域宽。为了保证转换得的数（即字符串表示的数）有足够的宽度，在必要时应在数的左边补上一定的空格。

3.7 break语句与continue语句

在循环语句执行过程中，除了通过测试从循环语句的顶部或底部正常退出外，有时从循环中直接退出来要显得更为方便一些。 break语句可用于从for、while与do-while语句中提前退出来，正如它可用于从 switch语句中提前退出来一样。 break语句可以用于立即从最内层的循环语句或 switch语句中退出。

下面的函数trim用于删除一个字符串尾部的空格符、制表符与换行符。它用了一个 break语句在找到最右边的非空格符、非制表符、非换行符时从循环中退出。

```
/* trim：删除字符串尾部的空格符、制表符与换行符 */
int trim(char s[]);
{
   int  n;

   for  (n = strlen(s)-1; n >= 0; n--)
        if  (s[n] != ' ' && s[n] != '\t' && s[n] != '\n')
            break;
   s[n+1] = '\n';
return  n;
}
```

strlen用于返回字符串的长度。 for循环语句用于从字符串的末尾反过来向前寻找第一个既不是空格符、制表符，也不是换行符的字符。循环在找到这样一个字符时中止执行，或在循环控制变量n变成负数时（即整个字符串都被扫描完时）终止执行。读者可以验证，即使是在字符串

为空或仅包含空白字符时，该函数也是正确的。

continue语句与break语句相关，但较少用到。continue语句用于使其所在的for、while或do-while语句开始下一次循环。在while与do-while语句中，continue语句的执行意味着立即执行测试部分；在for循环语句中，continue语句的执行则意味着使控制传递到增量部分。continue语句只能用于循环语句，不能用于switch语句。如果某个continue语句位于switch语句中，而后者又位于循环语句中，那么该continue语句用于控制下一次循环。

例如，下面这个程序段用于处理数组a中的非负元素。如果某个元素的值为负，那么跳过不处理。

```
for  (i =0; i < n; i++)  {
     if  (a[i] < 0)   /* 跳过负元素 */
         continue;
     …      /* 处理正元素 */
}
```

在循环的某些部分比较复杂时常常要使用continue语句。如果不使用continue语句，那么就可能要把测试反过来，或嵌入另一层循环，而这又会使程序的嵌套更深。

3.8 goto语句与标号

C语言提供了可以毫无节制使用的goto语句以及标记goto语句所要转向的位置的标号。从理论上讲，goto语句是没有必要的，实际上，不用它也能很容易地写出代码。本书即未使用goto语句。

然而，在有些情况下使用goto语句可能比较合适。最常见的用法是在某些深度嵌套的结构中放弃处理，例如一次中止两层或多层循环。break语句不能直接用于这一目的，它只能用于从最内层循环退出。下面是使用goto语句的一个例子：

```
for  ( … )
      for  ( … )  {
           …
           if  (disaster)
           goto error;
      }
   …
error:
   清理操作
```

如果错误处理比较重要并且在好几个地方都会出现错误，那么使用这种组织就比较灵活方便。

标号的形式与变量名字相同，其后要跟一个冒号。标号可以用在任何语句的前面，但要与相应的goto语句位于同一函数中。标号的作用域是整个函数。

再看一个例子，考虑判定在两个数组a与b中是否具有相同元素的问题。一种可能的解决方法是：

```
    for (i = 0; i < n; i++)
      for (j = 0; j < m; j++)
          if (a[i] == b[j])
```

```
            goto  found;
   /* 没有找到相同元素 */
   …
found:
   /* 取一个满足a[i] ==b[j]的元素 */
   …
```

所有带有goto语句的程序代码都可以改写成不包含goto语句的程序，但这可能需要以增加一些额外的重复测试或变量为代价。例如，可将这个判定数组元素是否相同的程序段改写成如下形式：

```
found = 0;
for (i = 0; i < n && !found; i++)
    for (j = 0; j < m && !found; j++)
        if (a[i] == b[j])
           found = 1;
if (found)
   /* 取一个满足a[i-1] ==b[j-1]的元素 */
 …
else
   /* 没有找到相同元素 */
 …
```

除了以上介绍的几个程序段外，依赖于goto语句的程序段一般都比不使用goto语句的程序段难以理解与维护。虽然不特别强调这一点，但我们还是建议尽可能减少goto语句的使用。

第4章　函数与程序结构

函数用于把较大的计算任务分解成若干个较小的任务，使程序人员可以在其他函数的基础上构造程序，而不需要从头做起。一个设计得当的函数可以把具体操作细节对程序中不需要知道它们的那些部分隐藏掉，从而使整个程序结构清楚，减轻了因修改程序所带来的麻烦。

C语言在设计函数时考虑了效率与易于使用这两个方面。一个 C程序一般都由许多较小的函数组成，而不是只由几个比较大的函数组成。一个程序可以驻留在一个文件中，也可以存放在多个文件中。各个文件可以单独编译并与库中已经编译过的函数装配在一起。但我们不打算详细讨论这一编译装配过程，因为具体编译与装配细节在各个编译系统中各不相同。

ANSI C标准对C语言所做的最显著的修改是在函数说明与定义这两个方面。正如第 1章所述，C语言现在已经允许在说明函数时说明变元的类型。为了使函数说明与定义匹配， ANSI C标准对函数定义的语法也做了修改。故编译程序可以查出比以前更多的错误。而且，如果变元说明得当，那么程序可以自动地进行适当的类型强制转换。

ANSI C标准进一步明确了名字的作用域规则，尤其是它要求每一个外部变量只能有一个定义。初始化做得更一般化了：现在自动数组与结构都可以初始化。

C的预处理程序的功能也得到了增强。新的预处理程序所包含的条件编译指令（一种用于从宏变元建立带引号字符串的方法）更为完整，对宏扩展过程的控制更严格。

4.1　函数的基本知识

下面首先设计并编写一个程序，用于把输入中包含特定的“ 模式 ”或字符串的各行打印出来（这是UNIX程序grep的特殊情况）。例如，对如下一组文本行查找包含字母字符串“ ould ”的行：

```
Ah Love! could you and I with Fate conspire
To grasp this sorry Scheme of Things entire,
Would not we shatter it to bits -- and then
Re-mould it nearer to the Heart's Desire!
```

可以产生如下输出：

```
Ah Love! could you and I with Fate conspire
Would not we shatter it to bits -- and then
Re-mould it nearer to the Heart's Desire!
```

这个程序段可以清楚地分成三部分：

```
while  ( 还有未处理的行 )
    if  ( 该行包含指定的模式 )
        打印该行
```

虽然可以把所有这些代码都放在主程序 main中，但一个更好的方法是把每一部分设计成一

个独立的函数。分别处理三个较小的部分要比处理一个大的整体容易，因为这样可以把不相关的细节隐藏在函数中，从而减少了不必要的相互影响的机会。而且这些函数也可以在其他程序中使用。

我们用函数getline来实现“还有未处理的行”，这个函数已在第1章介绍过；用printf函数来实现“打印该行”，这是一个别人早就为我们提供的函数，这意味着我们只需编写一个判定“该行包含指定的模式”的函数。

我们可以通过编写一个函数strindex(s, t)来解决这个问题，该函数返回字符串t在字符串s中出现的开始位置或位标，但当s中不包含t时，返回值为-1。由于C语言数组的下标从0开始，下标的值为0或正数，故用-1之类的负数作为失败信号是比较方便的。若以后需要更复杂的模式匹配，只需替换掉strindex函数即可，程序的其余部分可保持不动。（标准库中提供的库函数strstr的功能类似于strindex函数，只是该库函数返回的是指针而不是下标值。

在做了这样的设计后，填写程序的细节就比较简单了。下面即整个程序，读者可以看看各个部分是怎样组合在一起的。我们现在所要查找的模式是字面值字符串，它不是一种最通用的机制。我们将简单讨论一下字符数组的初始化方法，第5章将介绍如何在模式中加入可在程序运行时设置的参数。getline函数的版本也稍有不同，读者可将其与第1章所介绍的版本进行比较。

```
#include  <stdio.h>
#define  MAXLINE  100   /*最大输入行长度 */

int getline (char line[ ], int max);
int strindex(char source[ ], char searchfor[ ]);

char pattern[] = "ould";     /*要查找的模式 */

/* 找出所有与模式匹配的行 */
main ( )
{
   char  line[MAXLINE];
   int  found = 0;

   while  ( getline(line, MAXLINE) > 0 )
      if  ( strindex(line, pattern) >= 0 )  {
            printf( "%s", line);
            found++;
      }
   return  found;
}
/* getline：取一行放到s中，并返回该行的长度 */
int getline(char s[ ], int lim)
{
   int  c, i;

   i = 0;
   while  ( -- lim > 0&& ( c = getchar() ) != EOF && c != '\n' )
```

```
        s[i++] = c;
    if  (c == '\n' )
        s[i++] = c;
    s[i] = '\0';
    return  i;
}

/* strindex：返回t在s中的位置，若未找到则返回-1 */
int strindex(char s[], char t[] )
{
    int  i, j, k;
    for  ( i = 0; s[i] != '\0'; i++ )  {
           for  ( j =i, k = 0; t[k] != '\0' && s[j] ==t[k]; j++, k++ )
                ;
    if  ( k > 0 && t[k] == '\0' )
          return  i;
    }
    return  -1;
}
```

每一个函数定义均具有如下形式：

```
返回类型 函数名（变元说明表）
{
    说明序列与语句序列
}
```

函数定义的各个部分都可以缺省。最简单的函数结构如下：

```
dummy( ) { }
```

这个函数什么也不做、什么也不返回。像这种什么也不做的函数有时很有用，它可以在程序开发期间用做占位符。如果在函数定义中省略了返回类型，则缺省为 int。

程序是变量定义和函数定义的集合。函数之间的通信可以通过变元、函数返回值以及外部变量进行。函数可以以任意次序出现在源文件中。源程序可以分成多个文件，只要不把一个函数分在几个文件中就行。

return语句用于从被调用函数向调用者返回值， return之后可以跟任何表达式：

```
return 表达式；
```

在必要时要把表达式转换成函数的返回类型（结果类型）。表达式两边往往要加一对圆括号，但不是必需的，而是可选的。

调用函数可以随意忽略掉返回值。而且， return之后也不一定要跟一个表达式。在 return之后没有表达式的情况下，不向调用者返回值。当被调用函数因执行到最后的右花括号而完成执行时，控制同样返回调用者（不返回值）。如果一个函数在从一个地方返回时有返回值而从另一个地方返回时没有返回值，那么这个函数不一定非法，但可能存在问题。在任何情况下，如果一个函数不能返回值，那么它的“值”肯定是没有用的。

上面的模式查找程序从主程序 main中返回一个状态，即所匹配的字符串的数目。这个值可

以在调用该程序的环境中使用。

在不同系统上对驻留在多个源文件中的 C程序的编译与载入机制有很大的区别。例如，在UNIX系统上是用在第 1章中已提到过的 cc命令来完成这一任务的。假定有三个函数分别存放在名为main.c、getline.c与strindex.c的三个文件中，那么命令

```
cc  main.c getline.c strindex.c
```

用于编译这三个文件，并把目标代码分别存放在文件 main.o、getline.o与strindex.o中，然后再把这三个文件一起载入到可执行文件 a.out中。如果源程序中出现了错误（比如文件 main.c中出现了错误），那么可以用命令

```
cc  main.c getline.o strindex.o
```

对main.c文件重新编译，并将编译的结果与以前已编译过的目标文件 getline.o和strindex.o一起载入。cc命令用 .c 与 .o 这两种扩展名来区分源文件与目标文件。

练习4-1　编写一个函数strrindex(s, t)，用于返回字符串 t在s中最右出现的位置，如果 s中不包含t，那么返回 -1。

4.2 返回非整数值的函数

到目前为止，我们所讨论的函数均是不返回任何值（ void）或只返回int类型的值。假如一个函数必须返回其他类型的值，那么该怎么办呢？许多数值函数（如 sqrt、sin与cos等函数）返回的是double类型的值，另一些专用函数则返回其他类型的值。

为了说明让函数返回非整数值的方法，编写并使用函数 atof(s)，它用于把字符串 s转换成相应的双精度浮点数。 atof函数是atoi函数的扩充，第 2章与第3章已讨论了 atoi函数的几个版本。atof函数要处理可选的符号与小数点以及整数部分与小数部分。我们这个版本并不是一个高质量的输入转换函数，它所占用的空间比我们可以使用的要多。标准库中包含了具有类似功能的 atof函数，它在头文件<stdlib.h>中说明。

首先，由于atof函数返回值的类型不是 int，因此在该函数中必须说明它所返回值的类型。返回值类型的名字要放在函数名字之前：

```
#include <ctype.h>

/* 把字符串s转换成相应的双精度浮点数 */
double atof( char s[ ])
{
    double  val, power;
    int  i, sign;

    for  ( i = 0; isspace(s[i]); i++ )     /* 跳过空白 */
        ;
    sign = (s[i] == '-' ) ? -1 : 1;
   if ( s[i] == '+' || s[i] == '-' )
        i++;
    for  (val = 0.0; isdigit(s[i]); i++)
```

```
        val = 10.0 * val +(s[i] -'0' );
    if  (s [i] ] = = '.')
        i++;
    for  ( power = 1.0; isdigit(s[i]); i++)  {
        val = 10.0 * val +(s[i] -'0' );
        power *= 10.0;
    }
    return  sign * val / power;
}
```

其次，也是比较重要的，调用函数必须知道 atof函数返回的是非整数值。为了保证这一点，一种方法是在调用函数中显式说明atof函数。下面所示的基本计算器程序（仅适用于支票簿计算）中给出了这个说明，程序一次读入一行数（一行只放一个数，数的前面可能有一个正负号），并把它们加在一起，在每一次输入后把这些数的连续和打印出来：

```
#include  <stdio.h>

#define  MAXLINE  100

/* 基本计算器程序 */
main ( )
{
    double  sum, atof ( char [ ] );
    char  line[MAXLINE];
    int  getline(char line[], int max);

    sum =0;
    while  ( getline(line, MAXLINE) > 0 )
        printf( "\t%g\n", sum += atof(line) );
    return  0;
}
```

其中，说明语句

```
double  sum, atof ( char [ ] );
```

表明sum是一个double类型的变量，atof是一个具有char[ ]类型的变元且返回值类型为double的函数。

函数atof的说明与定义必须一致。如果 atof函数与调用它的主函数 main放在同一源文件中，并且具有不一致的类型，那么编译程序将会检测出这个错误。但是，如果 atof函数是独立编译的（这是一种更可能的情况），那么这种不匹配的错误就不会被检测出来， atof函数将返回double类型的值，而main函数则将之处理为int类型，从而这样所求得的结果毫无意义。

按照上述说明与定义匹配的讨论，这似乎很令人吃惊。发生不匹配现象的一个原因是，如果没有函数原型，则该函数在第一次出现的表达式中隐式说明，例如下面的表达式：

```
sum += atof(line)
```

如果在前面已经说明过的某个名字出现在某个表达式中并且左边跟一个左圆括号，那么就根据上下文认为该名字是函数名字，该函数的返回值类型为 int，但对变元没有给出上面信息。而且，

如果一个函数说明中不包含变元，比如

```
double atof ( );
```

那么也认为没有给出atof函数的变元信息，所有参数检查都被关闭。对空变元表做这种特殊的解释是为了使新的编译程序能编译比较老的C程序。但是，在新程序中也如此做是不明智的。如果一个函数有变元，那么说明它们；如果没有变元，那么使用 void。

借助恰当说明的atof函数，可以编写出函数atoi（将字符串转换成整数）：

```
/* atoi：利用atof函数把字符串s转换成整数 */
int atoi( char s[ ] )
{
   double  atof(char s[ ]);

   return  (int) atof ( s );
}
```

请注意其中说明和return语句的结构。在return语句：

```
return  表达式；
```

中的表达式的值在返回之前被转换成所在函数的类型。因此，如果对 atof函数的调用直接出现在atoi函数中的return语句中，如

```
return  atof ( s );
```

那么，由于函数atoi的返回值类型为int，系统要把atof函数的double类型的结果返回值自动转换成int类型。然而，这种操作可能会丢失信息，有些编译程序可能会为此给出警告信息。在此函数中由于采用了强制转换的方法显式地表明了所要做的转换操作，可以屏蔽有关警告信息。

练习4-2　对atof函数进行扩充，使之可以处理形如

```
123.45e-6
```

一类的科学表示法，即在浮点数后跟e或E与一个（可能有正负号的）指数。

4.3　外部变量

C程序由一组外部对象（外部变量或函数）组成。形容词 external与internal相对，internal用于描述定义在函数内部的函数变元以及变量。外部变量在函数外面定义，故可以在许多函数中使用。由于C语言不允许在一个函数中定义其他函数，因此函数本身是外部的。在缺省情况下，外部变量与函数具有如下性质：所有通过名字对外部变量与函数的引用（即使这种引用来自独立编译的函数）都是引用的同一对象（标准中把这一性质叫做外部连接）。在这个意义上，外部变量类似于FORTRAN语言的COMMON块或Pascal语言中在最外层分程序中说明的变量。后面将介绍如何定义只能在某个源文件使用的外部变量与函数。

由于外部变量是可以全局访问的，这就为在函数之间交换数据提供了一种可以代替函数变元与返回值的方法。任何函数都可以用名字来访问外部变量，只要这个名字已在某个地方做了说明。

如果要在函数之间共享大量的变量，那么使用外部变量要比使用一个长长的变元表更方便、

有效。然而，正如在第1章所指出的，这样使用必须充分小心，因为这样可能对程序结构产生不好的影响，而且可能会使程序在各个函数之间产生太多的数据联系。

外部变量的用途还表现在它们比内部变量有更大的作用域和更长的生存期。自动变量只能在函数内部使用，当其所在函数被调用时开始存在，当函数退出时消失。而外部变量是永久存在的，它们的值在从一次函数调用到下一次函数调用之间保持不变。因此，如果两个函数必须共享某些数据，而这两个函数都互不调用对方，那么最为方便的是，把这些共享数据作成外部变量，而不是作为变元来传递。

下面通过一个更大的例子来说明这个问题。问题是要编写一个具有加（+）、减（-）、乘（*）、除（/）四则运算功能的计算器程序。为了更易于实现，在计算器中使用逆波兰表示法来代替普通的中辍表示法（逆波兰表示法用在某些袖珍计算器中，诸如Forth与Postscript等语言也使用了逆波兰表示法）。

在使用逆波兰表示法时，所有运算符都跟在其运算分量的后面。诸如

```
(1 - 2) * (4 + 5)
```

一类的中辍可用逆波兰表示法表示成：

```
1 2 - 4 5 + *
```

在使用逆波兰表示法时不再需要圆括号，只需知道每一个运算符需要几个运算分量。

计算器程序的实现很简单。每一个运算分量都被依次下推到栈中；当一个运算符到达时，从栈中弹出相应数目的运算分量（对二元运算符是两个运算分量），把该运算符作用于所弹出的运算分量，并把运算结果再下推回栈中。例如，对上面所述逆波兰表达式，首先把1与2下推到栈中，再用两者之差-1来取代它们；然后，把4与5下推到栈中，再用两者之和9来取代它们。最后，从栈中取出栈顶的-1与9，把它们的积-9下推到栈顶。当到达输入行的末尾时，把栈顶的值弹出并打印出来。

这样，该程序的结构是一个循环，每一次循环对一个运算符及相应的运算分量执行一次操作：

```
while  ( 下一个运算符或运算分量不是文件结束指示符 )
    if  ( 数 )
        将该数下推到栈中
    else if  ( 运算符 )
        弹出所需数目的运算分量
        执行运算
        将结果下推到栈中
    else if  ( 换行符 )
        弹出并打印栈顶的值
    else
        错误
```

栈的下推与弹出操作比较简单，但是，如果把错误检测与恢复操作都加进去，那么它们就会显得很长，最好把它们设计成独立的函数，而不要把它们作为在这个程序中重复的代码段。另外还需要一个单独的函数来取下一个输入运算符或运算分量。

到目前为此还没有讨论的主要设计决策是，把栈放在哪里？即哪些函数可以直接访问它？

一种可能是把它放在主函数 main中，把栈及其当前位置作为传递给要对它进行下推或弹出弹出操作的函数。但是， main函数不需要知道控制该栈的变量信息，它只进行下推与弹出操作。因此，可以把栈及其相关信息放在外部变量中，并只供 push与pop函数访问，而不能为main函数则所访问。

把上面这段话翻译成代码很容易。如果把这个程序放在一个源文件中，那么它为如下形式：

```
#include …
#define …

用于main的函数说明

main() { … }

用于push与pop的外部变量

void push ( double f ) { … }
double pop(void) { … }

int getop(char s[ ]) { … }

被getop调用的函数
```

我们在以后将讨论怎样把这个程序分割成两个或多个源文件。

main函数主要由一个循环组成，该循环中包含了一个对运算符与运算分量进行分情形操作的switch语句，这里对switch语句的使用要比3.4节所示的例子更为典型。

```
#include  <stdio.h>
#include  <stdlib.h>        /* 供atof()函数使用 */

#define  MAXOP   100        /* 运算分量或运算符的最大大小 */
#define  NUMBER  '0'        /* 表示找到数的信号 */

int getop ( char [ ] );
void push ( double f );
double pop(void);

/* 逆波兰计算器 */
main ( )
{
   int  type;
   double  op2;
   char  s[MAXOP];

   while  ( ( type = getop(s) ) != EOF )  {
      switch  ( type )  {
      case  NUMBER:
         push(atof(s));
         break;
```

```
        case  '+':
           push ( pop() + pop());
           break;
        case  '*':
           push ( pop() * pop());
           break;
        case  '-':
           op2 = pop( );
           push ( pop() - op2);
           break;
        case  '/':
           op2 = pop( );
           if  ( op2 != 0 )
               push ( pop() / op2 );
        else
               printf ( "error: zero divisor\n" );
           break;
        case  '\n':
           printf ( "\t%.8g\n", pop( ) );
           break;
        default:
           printf ( "error: unknown command %s\n", s);
           break;
        }
     }
     return  0;
}
```

由于 + 与 * 是两个满足交换律的运算符，因此弹出的两个运算分量的次序无关紧要，但是，- 与 / 的左右运算分量的次序则是必需的。在如下所示的函数调用中：

```
push ( pop() - pop() );     /* 错 */
```

对pop函数的两次调用的次序没有定义。为了保证正确的次序，必须像在 main函数中一样把其第一个值弹出到一个临时变量中。

```
#define  MAXVAL  100      /* 栈val的最大深度  */

int  sp = 0;              /* 下一个自由栈元素位置 */
double  val[MAXVAL];      /* 值栈 */

/* push：把f下推到值栈中 */
void push ( double f )
{
   if  ( sp < MAXVAL )
       val[sp++] = f;
   else
       printf ( "error: stack full, can't push %g\n", f);
}
```

```
/* pop：弹出并返回栈顶的值 */
double pop(void);
{
   if ( sp > 0 )
       return val[--sp];
   else  {
       printf ( "error: stack empty\n" );
       return  0.0;
   }
}
```

一个变量如果在函数的外面定义，那么它就是外部变量。因此，我们把必须为 push和pop函数共享的栈和栈顶指针定义在这两个函数的外面。但 main函数本身并没有引用该栈或栈顶指针，因此将它们对它隐藏。

下面讨论getop函数的实现，它用于取下一个运算符或运算分量。这一任务比较容易。跳过空格与制表符。如果下一个字符不是数字或小数点，那么返回；否则，把这些数字字符串收集起来（其中可能包含小数点），并返回NUMBER，用这个信号表示数已经收集起来了。

```
#include  <ctype.h>

int getch(void);
void ungetch(int);

/* getop：取下一个运算符或数值运算分量 */
int getop(char s[ ] )
{
   int  i, c;

   while  ( (s[0] = c = getch() ) == ' ' || c == '\t' )
    ;
   s[1] = '\0';
   if  ( !isdigit( c ) && c != '.' )
       return c;     /* 不是数 */
       i = 0;
   if  ( isdigit( c ) )  /* 收集整数部分*/
       while  ( isdigit(s[++i] = c = getch( ) ) )
           ;
   if  (c == '.' )     /* 收集小数部分 */
       while  ( isdigit(s[++i] = c = getch( ) ) )
           ;
   s[i] = '\0';
   if  (c != EOF )
       ungetch( c );
   return  NUMBER;
}
```

这段程序中的getch与ungetch是两个什么样的函数呢？在程序中经常会出现这样的情况，一个程序在读进过多的输入之前不能确定它已经读入的输入是否足够。一个例子是在读进用于组

成数的字符的时候：在看到第一个非数字字符之前，所读入数的完整性是不能确定的。由于程序要超前读入一个字符，最后有一个字符不属于当前所要读入的数。

如果能“反读”不需要的字符，那么这个问题就能得到解决。每当程序多读进一个字符时，就可以把它推回到输入中，对代码其余部分而言就像这个字符并没有读过一样。我们可以通过编写一对相互配合的函数来比较方便地模拟反取字符操作。getch函数用于读入下一个待处理的字符，而ungetch函数则用于把字符放回到输入中，使得此后对getch函数的调用将在读新的输入之前先返回经ungetch函数放回的那些字符。

把这两个函数放在一起配合使用很简单。ungetch函数把要推回的字符放到一个共享缓冲区（字符数组）中，而getch函数在该缓冲区不空时就从中读取字符，在缓冲区为空时调用getchar函数直接从输入中读字符。为了记住缓冲区中当前字符的位置，还需要一个下标变量。

由于缓冲区与下标变量是供getch与ungetch函数共享的，在两次调用之间必须保持值不变，它们必须作成这两个函数的外部变量。这样，可以如下编写getch与ungetch函数及其共享变量：

```
#define  BUFSIZE  100

char  buf[BUFSIZE];      /* 用于unget函数的缓冲区 */
int  bufp = 0;           /* buf中下一个自由位置 */

int getch(void)          /* 取一个字符（可能是推回的字符） */
{
return  ( bufp > 0 ) ? buf[--bufp] : getchar( );
}

void ungetch(int c)       /* 把字符推回到输入中 */
{
   if  ( bufp >= BUFSIZE )
       printf ( "ungetch: too many characters\n" );
   else
       buf[bufp++] = c;
}
```

标准库中提供了函数ungetc，用于推回一个字符，第7章将对它进行讨论。为了说明更一般的方法，我们这里使用了一个数组而不是一个字符用于推回字符。

练习4-3　在有了基本框架后，对计算器程序进行扩充就比较简单了。在该程序中加入取模（%）运算符并注意负数的情况。

练习4-4　在栈操作中添加几个命令分别用于在不弹出时打印栈顶元素、复制栈顶元素以及交换栈顶两个元素的值。再增加一个命令用于清空栈。

练习4-5　增加对诸如sin、exp与pow等库函数的访问操作。有关这些库函数参见附录B.4节中的头文件<math.h>。

练习4-6　增加处理变量的命令（提供26个由单字母变量很容易）。增加一个变量存放用于最近打印的值。

练习4-7　编写一个函数ungets(s)，用于把整个字符串推回到输入中。ungets函数要使用buf与bufp吗？它可否仅使用ungetch函数？

练习4-8　假定最多只要推回一个字符。请相应地修改getch与ungetch这两个函数。

练习4-9　上面所介绍的getch与ungetch函数不能正确地处理推回的EOF。决定当推回EOF时应具有什么性质，然后再设计实现。

练习4-10　另一种组织方法是用getline函数读入整个输入行，这样便无需使用getch与ungets函数。运用这一方法修改计算器程序。

4.4 作用域规则

用以构成C程序的函数与外部变量完全没有必要同时编译，一个程序可以放在几个文件中，可以从库中调入已编译过的函数。我们比较感兴趣的问题主要有：

- **怎样编写说明才能使所说明的变量在编译时被认为是正确的？**
- **怎样安排说明才能保证在程序载入时各部分能正确相连？**
- **怎样组织说明才能使得只需一份拷贝？**
- **怎样初始化外部变量？**

为了便于讨论这些问题，我们把计算器程序组织在若干个文件中。从实用角度看，计算器程序比较小，不值得分几个文件存放，但通过它可以很好地说明在较大的程序中所遇到的有关问题。

一个名字的作用域指程序中可以使用该名字的部分。对于在函数开头说明的自动变量，其作用域是说明该变量名字的函数。在不同函数中说明的具有相同名字的各个局部变量毫不相关。对于函数的参数也如此，函数参数实际上可以看作是局部变量。

外部变量或函数的作用域从其说明处开始一直到其所在的被编译的文件的末尾。例如，如果main、sp、val、push与pop是五个依次定义在某个文件中的函数与外部变量，即：

```
main( )  { … }

int  sp = 0;
double  val[MAXVAL];

void push( double f )  { … }

double pop( void )  { … }
```

那么，在push与pop这两个函数中不需做任何说明就可以通过名字来访问变量sp与val，但是，这两个变量名字不能用在main函数中，push与pop函数也不能用在main函数中。

另一方面，如果一个外部变量在定义之前就要使用到，或者这个外部变量定义在与所要使用它的源文件不相同的源文件中，那么要在相应的变量说明中强制性地使用关键词extern。

将对外部变量的说明与定义严格区分开来很重要。变量说明用于通报变量的性质（主要是变量的类型），而变量定义则除此以外还引起存储分配。如果在函数的外部包含如下说明：

```
int  sp;
double  val[MAXVAL];
```

那么这两个说明定义了外部变量 sp与val，并为之分配存储单元，同时也用作供源文件其余部分使用的说明。另一方面，如下两行:

```
extern int  sp;
extern double  val[MAXVAL];
```

为源文件剩余部分说明了 sp是一个int类型的外部变量， val是一个double数组类型的外部变量(该数组的大小在其他地方确定)，但这两个说明并没有建立变量或为它们分配存储单元。

在一个源程序的所有源文件中对一个外部变量只能在某个文件中定义一次，而其他文件可以通过extern说明来访问它(在定义外部变量的源文件中也可以包含对该外部变量的 extern说明)。在外部变量的定义中必须指定数组的大小，但在 extern说明中则不一定要指定数组的大小。

外部变量的初始化只能出现在其定义中。

假定函数push与pop在一个文件中定义，变量 val与sp在另一个文件中定义并初始化(虽然一般不可能这样组织程序)。这些定义与说明必须把这些函数和变量捆在一起：

在文件file1中：

```
   extern int  sp;
   extern double  val [ ];

   void push( double f )  { … }

   double pop( void )  { … }
```

在文件file2中：

```
   int  sp = 0;
   double  val[MAXVAL];
```

由于文件 file1中的extern说明不仅放在函数定义的外面而且还放在它们前面，故它们适用于所有函数，这一组说明对文件 file1已足够了。如果 sp与val的定义跟在对它们的使用之后，那么也要这样来组织文件。

4.5 头文件

下面考虑把计算器程序分成若干个源文件。主函数 main单独放在文件main.c中，push与pop函数及它们所使用的外部变量放在第二个文件 stack.c中，getop函数放在第三个文件 getop.c中，getch与ungetch函数放在第四个文件 getch.c中。之所以把它们分开，是因为在实际程序中它们来自于一个独立编译的库。

还有一个问题需要考虑，即这些文件之间的定义与说明的共享问题。我们将尽可能使所要共享的部分集中在一起，以使得只需一个拷贝，当要对程序进行改进时也能保证程序的正确性。我们将把这些公共部分放在头文件 calc.h中，在需要使用该头文件时可以用 #include指令引入(#include指令将在4.11节介绍)。如此得到的程序形式如下所示：

头文件calc.h：

```
#define  NUMBER  '0'

void push( double );
double pop( void );
int getop( char [ ] );
int getch( void);
void ungetch( int );
```

文件main.c：

```
#include  <stdio.h>
#include  <stdlib.h>
#include  "calc.h"
#define  MAXOP  100

main( )
{
    …
}
```

文件getop.c：

```
#include  <stdio.h>
#include  <ctype.h>
#include  "calc.h"
getop( )
{
    …
}
```

文件getch.c：

```
#include  <stdio.h>
#define  BUFSIZE  100
char  buf[BUFSIZE];
int  bufp = 0;
int getch( void )
{
    …
}
void ungetch( int)
{
    …
}
```

文件stack.c：

```
#include  <stdio.h>
#include  "calc.h"
#define  MAXVAL  100
int  sp = 0;
```

```
double  val[MAXVAL];

void push( double )
{
    …
}

double pop( void)
{
    …
}
```

我们对如下两个方面做了折衷：一方面是对每一个文件只能访问它完成任务所需要的信息的要求，另一方面是维护较多的头文件比较困难的现实。对于某些中等规模的程序，最好是只使用一个头文件来存放程序中各个部分需要共享的实体，这是我们在这里所做的结论。对于比较大的程序，需要做更精心的组织，使用更多的头文件。

4.6 静态变量

stack.c文件中定义的变量sp与val以及getch.c文件中定义的变量buf 与bufp仅供它们各自所在的源文件中的函数使用，不能被其他函数访问。 static说明适用于外部变量与函数，用于把这些对象的作用域限定为被编译源文件的剩余部分。通过外部 static对象，可以把诸如buf与bufp一类的名字隐藏在getch-ungetch组合中，使得这两个外部变量可以被 getch与ungetch函数共享，但不能被getch与ungetch函数的调用者访问。

可以在通常的说明之前前缀以关键词 static来指定静态存储。如果把上述两个函数与两个变量放在一个文件中编译，如下：

```
static char  buf[BUFSIZE];     /* 供ungetch函数使用的缓冲区 */
static int  bufp = 0;          /* 缓冲区buf的下一个自由位置 */

int getch( void )  { …}

void ungetch( int c)  { … }
```

那么其他函数不能访问变量 buf与bufp，做这两个名字不会和同一程序中其他文件中的同名名字相冲突。基于同样的理由，可以通过把变量 sp与val说明为静态的，使这两个变量只能供进行栈操作的push与pop函数使用，而对其他文件隐藏。

外部static说明最常用于说明变量，当然它也可用于说明函数。通常情况下，函数名字是全局的，在整个程序的各个部分都可见。然而，如果把一个函数说明成静态的，那么该函数名字就不能用在除该函数说明所在的文件之外的其他文件中。

static说明也可用于说明内部变量。内部静态变量就像自动变量一样局部于某一特定函数，只能在该函数中使用，但与自动变量不同的是，不管其所在函数是否被调用，它都是一直存在的，而不像自动变量那样，随着所在函数的调用与退出而存在与消失。换而言之，内部静态变量是一种只能在某一特定函数中使用的但一直占据存储空间的变量。

练习4-11　修改getop函数，使之不再需要使用 ungetch函数。提示：使用一个内部静态变量。

4.7 寄存器变量

register说明用于提醒编译程序所说明的变量在程序中使用频率较高。其思想是，将寄存器变量放在机器的寄存器中，这样可以使程序更小、执行速度更快。但编译程序可以忽略此选项。

register说明如下所示：

```
register int  x;
register char  c;
```

寄存器说明只适用于自动变量以及函数的形式参数。对于后一种情况，例于如下：

```
f( register unsigned m, register long n )
{
    register int  i;
    …
}
```

在实际使用时，由于硬件环境的实际情况，对寄存器变量会有一些限制。在每一个函数中只有很少的变量可以放在寄存器中，也只有某些类型的变量可以放在寄存器中。然而，过量的寄存器说明并没有什么害处，因为对于过量的或不允许的寄存器变量说明，编译程序可以将之忽略掉。另外，不论一个寄存器变量实际上是不是存放在寄存器中，它的地址都是不能访问的（关于这一问题将在第5章讨论）。对寄存器变量的数目与类型的具体限制视不同的机器而有所不同。

4.8 分程序结构

C语言不是Pascal等语言意义上的分程序结构的语言，因为它不允许在函数中定义函数。但另一方面，变量可以以分程序结构的形式在函数中定义。变量的说明（包括初始化）可以跟在用于引入复合语句的左花括号的后面，而不是只能出现在函数的开始部分。以这种方式说明的变量可以隐藏在该分程序外面说明的同名变量，并在与该左花括号匹配的右花括号出现之前一直存在。例如，在如下程序段中：

```
if  ( n > 0 )  {
   int  i;     /* 说明一个新的i */

   for  ( i = 0; i < n; i++ )
    …
}
```

变量i的作用域是if语句的“真”分支，这个i与在该分程序之外说明的i无关。在分程序中说明与初始化的自动变量每当进入这个分程序时就被初始化。静态变量只在第一次进入分程序时初始化一次。

自动变量（包括形式参数）也隐藏同名的外部变量与函数。对于如下说明：

```
int  x;
```

```
int  y;

f ( double x )
{
    double  y;
    …
}
```

在函数f内，所出现的x引用的是参数，其类型为double，而在函数f之外，引用的是类型为int的外部变量。对变量y也如此。

就风格而言，最好避免出现变量名字隐藏外部作用域中同名名字的情况，否则可能会出现大量混乱与错误。

4.9 初始化

前面已多次提到初始化的概念，但一直没有认真讨论它。这一节在前面讨论了各种存储类的基础上总结一些初始化规则。

在没有显式初始化的情况下，外部变量与静态变量都被初始化为 0，而自动变量与寄存器变量的初值则没有定义（即，其初值是“垃圾”）。

在定义纯量变量时，可以通过在所定义的变量名字后加一个等号与一个表达式来进行初始化：

```
int   x = 1;
char  squote = '\'';
long  day = 1000L * 60L *60L * 24L;   /* 每天的毫秒数 */
```

对于外部变量与静态变量，初始化符必须是常量表达式，初始化只做一次（从概念上讲是在程序开始执行前进行初始化）。对于自动变量与寄存器变量，则在每当进入函数或分程序时进行初始化。

对于自动变量与寄存器变量，初始化符不一定限定为常量：它可以是任何表达式，其中可以包含前面已定义过的值甚至可以包含函数调用。对 3.3节介绍的二分查找程序的初始化可以用如下形式：

```
int binsearch( int x, int v[ ], int n )
{
   int  low = 0;
   int  high = n - 1;
   int  mid;
   …
}
```

来代替原来的形式：

```
int low, high, mid;

low = 0;
high = n - 1;
```

实际上，自动变量的初始化部分就是赋值语句的缩写。到底使用哪一种形式还是一个尚待

尝试的问题，我们一般使用显式的赋值语句，因为说明中的初始化符比较难以为人们发现，并且距使用点比较远。

数组的初始化也是通过说明中的初始化符完成的。数组初始化符是用花括号括住并用逗号分隔的初始化符序列。例如，当要用每一个月的天数来初始化数组 days时，可用如下变量定义：

```
int days[ ] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
```

当数组的大小缺省时，编译程序就通过统计花括号中初始化符的个数作为数组的长度，本例中数组的大小为12个元素。

如果初始化符序列中初始化符的个数比数组元素数少，那么对于没有得到初始化的数组元素在该数组为外部变量、静态变量与自动变量时被初始化为 0。如果初始化符序列中初始化符的个数比数组元素数多，那么就是错误的。我们既无法一次性地为多个数组元素指定一个初始化符，也不能在没有指定前面数组元素值的情况下初始化后面的数组元素。

字符数组的初始化比较特殊，可以用一个字符串来代替用花括号括住并用逗号分隔的初始化符序列：

```
char  pattern [] = "ould";
```

它是如下虽然长些但却与之等价的定义的缩写：

```
char  pattern [] = { 'o', 'u', 'l', 'd' , '\0' };
```

在此情况下，数组的大小是5（4个字符外加一个字符串结束符 '\0'）。

4.10 递归

C函数可以递归调用，即一个函数可以直接或间接调用自己。考虑把一个数作为字符串打印的情况。如前所述，数字是以相反的次序生成的：低位数字先于高位数字生成，但它们必须以相反的次序打印出来。

对这一问题有两种解决方法。一种方法是将生成的各个数依次存储到一数组中，然后再以相反的次序把它们打印出来，正如3.6节对itoa函数所做的那样。另一种方法是使用递归解法，用于完成这一任务的函数 printd首先调用自身处理前面的（高位）数字，然后再把后面的数字打印出来。这个版本不能处理最大的负数。

```
#include  <stdio.h>

/* printd：以十进制打印数n */
void printd(int n )
{
   if  ( n < 0 )  {
       putchar( '-' );
       n = -n;
   }
   if  ( n / 10 )
       printd( n / 10 );
   putchar( n % 10 + '0' );
}
```

当一个函数递归调用自身时，每一次调用都会得到一个与以前的自动变量集合不同的新的自动变量集合。因此，在调用 printd (123) 时，第一次调用printd的变元n = 123。它把12传递给对printd的第二次调用，后者又把 1传递给对printd的第三次调用。第三次对 printd的调用将先打印1，然后再返回到第二次调用。从第三次调用返回后的第二次调用同样先打印 2，然后再返回到第一次调用。后者打印出3后结束执行。

另一个用于说明递归的一个例子是快速排序。快速排序算法是 C. A. R. Hoare于1962年发明的。对于一个给定的数组，从中选择一个元素（叫做分区元素），并把其余元素划分成两个子集合 —— 一个是由所有小于分区元素的元素组成的子集合，另一个是由所有大于等于分区元素的元素组成的子集合。对这样两个子集合递归应用同一过程。当某个子集合中的元素数小于两个时，这个子集合不需要再排序，故递归停止。

下面这个版本的快速排序函数可能不是最快的一个，但它是最简单的一个。在每一次划分子集合时都选取各个子数组的中间元素。

```
/* qsort：以递增顺序对v[left] … v[right] 进行排序 */
void qsort( int v[], int left, int right )
{
   int  i, last;
   void swap( int v[], int i, int j );

   if  ( left >= right )    /* 若数组所包含的元素数少于两个，则什么也不做 */
       return;
   swap( v, left, (left + right)/2 );    /* 把分区元素移到v[0] */
   last = left;
   for  ( i = left+1; i <= right; i++ )    /* 分区 */
        if  ( v[i] < v[left] )
              swap( v, ++last, i );
   swap( v, left, right );    /* 恢复分区元素 */
   qsort( v, left, last-1 );
   qsort( v, last+1, right );
}
```

之所以把数组元素交换操作作为一个独立的函数 swap，是因为它在qsort函数中要使用三次。

```
/* swap：交换v[i]与v[j]的值 */
void swap( int v[], int i, int j )
{
   int  temp;

   temp = v[i];
   v[i] = v[j];
   v[j] = temp;
}
```

标准库中包含了qsort函数的一个版本，它可用于对任何类型的对象排序。

递归的中间结果不需在内存特别保存，因为在某处有一个被处理值的栈。递归的执行速度并不快，但递归代码比较紧致，要比相应的非递归代码易于编写与理解。在描述诸如树等递归

定义的数据结构时使用递归尤其方便，6.5节将给出这方面的一个例子。

练习4-12　运用printd函数的思想编写一个递归版本的itoa函数，即通过递归调用把整数转换成字符串。

练习4-13　编写一个递归版本的reverse(s)函数，把字符串s颠倒过来。

4.11　C预处理程序

C语言通过预处理程序提供了一些语言功能，预处理程序从理论上讲是编译过程中单独进行的第一个步骤。其中两个最常用的预处理功能是#include指令（用于在编译期间把指定文件的内容包含进当前文件中）与#define指令：（用任意字符序列取代一个标记）。本节还将介绍其他功能，如条件编译与带变元的宏。

4.11.1　文件包含

文件包含指令，即#include指令，使我们比较容易处理一组#define指令以及说明等。在源程序文件中，任何形如：

```
#include  "文件名"
```

或

```
#include  <文件名>
```

的行都被替换成由文件名所指定的文件的内容。如果文件名用引号括起来，那么就在源程序所在位置查找该文件；如果在这个位置没有找到该文件，或者如果文件名用尖括号<与>括起来，那么就按实现定义的规则来查找该文件。被包含的文件本身也可包含#include指令。

在源文件的开始处一般都要有一些#include指令，或包含#define语句与extern说明，或访问诸如<stdio.h>等头文件中库函数的函数原型说明。（严格地说，这些没有必要做成文件。访问头文件的细节依赖于实现。）

对于比较大的程序，#include指令是把各个说明捆在一起的优选方法。它使所有源文件都被提供以相同的定义与变量说明，从而可避免发生一些特别讨厌的错误。自然地，如果一个被包含的文件的内容做了修改，那么所有依赖于这个被包含文件的源文件都必须重新编译。

4.11.2　宏替换

宏定义，即#define指令，具有如下形式：

```
#define  名字  替换文本
```

它是一种最简单的宏替换——出现各个的名字都将被替换文本替换。#define指令中的名字与变量名具有相同的形式，替换文本可以是任意字符串。正常情况下，替换文本是#define指令所在行的剩余部分，但也可以把一个比较长的宏定义分成若干行，这时只需在尚待延续的行后加上一个反斜杠 \ 即可。#define指令所定义的名字的作用域从其定义点开始到被编译的源文件的结束。在宏定义中也可以使用前面的宏定义。替换只对单词进行，对括在引号中的字符串不起作用。例如，如果YES是一个被定义的名字，那么在printf ("YES") 或YESMAN中不能进行替换。

用替换文本可以定义任何名字，例如：

```
#define  forever  for ( ; ; )    /* 无限循环 */
```

为无限循环定义了一个新的关键词forever。

在宏定义中也可以带变元，这样可以对不同的宏调用使用不同的替换文本。例如，可通过：

```
#define  max( A, B )  ( ( A ) > ( B ) ? ( A ) : ( B ) )
```

定义一个宏max。对max的使用看起来很像是函数调用，但宏调用是直接将替换文本插入到代码中。形式参数（在此为A与B）的每一次出现都被替换成对应的实在变元。于是，语句

```
x = max( p+q, r+s );
```

将被替换成

```
x = ( ( p+q ) > ( r+s ) ? ( p+q ) : ( r+s ) );
```

只要变元能得到一致的处理，宏定义可以用于任何数据类型。没有必要像函数那样为不同数据类型定义不同的max。

如果仔细检查一下max的展开式，那么你将注意到它存在某些缺陷。其中作为变元的表达式要重复计算两次，当表达式中会带来副作用（如含有加一运算符或输入输出）时，会出现很坏的情况。例如，

```
max( i++, j++ )   /* 错 */
```

将对每一个变元做两次加一操作。同时也必须小心，为了保证计算次序的正确性要适当使用圆括号。请读者考虑一下，对于宏定义

```
#define  square( x )  x * x     /* 错 */
```

当用square(z + 1)调用它时会出现什么情况。

然而，宏还是很有价值的。在<stdio.h>头文件中有一个很实用的例子，getchar与putchar函数在实际上往往被定义为宏，这样可以避免在处理字符时调用函数所需的运行时开销。在<ctype.h>头文件中定义的函数也常常用宏来实现。

可以用#undef指令取消对宏名字的定义，这样做通常是为了保证一个调用所调用的是一个实际函数而不是宏：

```
#undef  getchar
int  getchar( void )  { … }
```

形式参数不能用带引号的字符串替换。然而，如果在替换文本中，参数名以#作为前缀，那么它们将被由实际变元替换的参数扩展成带引号的字符串。例如，可以将其与字符串连接运算结合起来制作调试打印宏：

```
#define  dprint( expr )  printf( #expr " = %g\n", expr )
```

当用诸如

```
dprint( x/y );
```

调用该宏时，该宏就被扩展成

```
printf( "x/y" " = %g\n", x/y );
```

其中的字符串被连接起来，即这个宏调用的效果是：

```
printf( "x/y = %g\n", x/y );
```

在实际变元中，双引号 " 被替换成 \"、反斜杠 \ 被替换成 \\，故替换后的字符串是合法的字符串常量。

预处理运算符 ##为宏扩展提供了一种连接实际变元的手段。如果替换文本中的参数用 ##相连，那么参数就被实际变元替换， ##与前后的空白字符被删除，并对替换后的结果重新扫描。例如，下面定义的宏paste用于连接两个变元：

```
#define  paste( front, back )  front ## back
```

从而宏调用paste(name, 1)的结果是建立单词name1。

关于##嵌套使用的规则比较难以掌握，详细细节请参阅附录 A。

练习4-14　定义宏swap(t, x, y)，用于交换t类型的两个变元（使用分程序结构）。

4.11.3　条件包含

在预处理语句中还有一种条件语句，用于在预处理中进行条件控制。这提供了一种在编译过程中可以根据所求条件的值有选择地包含不同代码的手段。

#if语句中包含一个常量整数表达式（其中不得包含sizeof、类型强制转换运算符或枚举常量），若该表达式的求值结果不等于 0时，则执行其后的各行，直到遇到 #endif、#elif或#else语句为止（预处理语句#elif类似于if语句的else if结构）。在#if语句中可以使用一个特殊的表达式 defined(名字)：当名字已经定义时，其值为 1；否则，其值为0。

例如，为了保证 hdr.h文件的内容只被包含一次，可以像下面这样用条件语句把该文件的内容包围起来：

```
#if  !defined( HDR )
#define  HDR

/* hdr.h文件的内容*/

#endif
```

被#if与#endif包含的第一行定义了名字HDR，其后的各行将会发现该名字已有定义并跳到#endif。还可以用类似的样式来避免多次重复包含同一文件。如果连续使用这种，那么每一个头文件中都可以包含它所依赖的其他头文件，而不需要它的用户去处理这种依赖关系。

下面的预处理语句序列用于测试名字 SYSTEM以确定要包含进哪一个版本的头文件：

```
#if SYSTEM == SYSV
    #define HDR  "sysv.h"
#elif SYSTEM == BSD
    #define HDR  "bsd.h"
#elif SYSTEM == MSDOS
    #define HDR  "msdos.h"
#else
    #define HDR  "default.h"
```

```
#endif
# include  HDR
```

当需要测试一个名字是否已经定义时，可以使用两个特殊的预处理语句： #ifdef与#ifndef。可以使用#ifdef将上面第一个关于#if的例子改写如下：

```
#ifdef  HDR
#define  HDR

/* hdr.h文件的内容*/

#endif
```